(12)

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 86-32-03	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) UW/NW VLSI Consortium Semiannual Technical Report No. #3		5. TYPE OF REPORT & PERIOD COVERED Technical, interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) UW/NW VLSI Consortium		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-K-0072 ARPA-4563, #2 Code 5D30
9. PERFORMING ORGANIZATION NAME AND ADDRESS UW/NW VLSI Consortium, Dept. of Computer Science University of Washington, FR-35 Seattle, WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA - ISTO 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE December, 1986 13. NUMBER OF PAGES 72
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR University of Washington 315 University District Building 1107 NE 45th St., JD-16, Seattle, WA 98195		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
VLSI Design Generators, CMOS, PLA, energy complexity, NC, CFL, MOS, VLSI Consortium

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This document reports on the research activities of the University of Washington/Northwest VLSI Consortium for the period of Marcy 18, 1986 to December 10, 1986 under sponsorship of the Defense Advanced Research Projects Agency, under contract number MDA903-85-K-0072, program code number 5D30.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UW/NW VLSI CONSORTIUM

Semiannual Technical Report No. 3

University of Washington

December 10, 1986
TR# 86-32-03

Reporting Period: 18 March 1986 to 10 December 1986

Principal Investigator: Lawrence Snyder

Sponsored by
Defense Advanced Research Projects Agency(DoD)
ARPA Order No. 4563/2
Issued by Defense Supply Service-Washington
Under Contract #MDA903-85-K-0072
(Program Code Number: 5D30)

Contents

1 Executive Summary

This document reports on the research activities of the UW/NW VLSI Consortium for the period 18 March 1986 to 10 December 1986 under sponsorship of the Defense Advanced Research Projects Agency. The applicable contract for this period is MDA 903-85-K-0072. On January 1, 1986 the UW/NW VLSI Consortium will change its name to the Northwest Laboratory for Integrated Systems.

One of the outstanding contributions during this reporting period has been our work in the energy complexity analysis of VLSI designs. An extensive theoretical analysis has been made of the energy requirements of several classes of functions, in particular transitive and one-switchable. A surprising result of this work is that, for systolic implementations of some of these functions, the average case energy consumption is of the same order as worst case. For this reason the energy optimization of many datapath systolic systems gives negligible returns. Another work in this area analyzes the speed and energy tradeoffs of dynamic and static PLA designs within various application domains.

A major project of this reporting period has been the definition of a model for generator construction. the intent of the model is to provide a concise representation that captures the fundamental structural and functional properties of the circuit. With this representation a variety of output descriptions may be derived.

During the past six months the model has been refined and a language parser written. Work is currently underway on the backend programs that analyze the model and produce schematics, layouts, and functional descriptions.

The importance of generating high quality circuits which are free of noise and metal migration problems has prompted us to investigate means of analyzing the current carrying capacities of power bus structures in MOS designs. We have begun work on a tool called Hercules that will perform such an analysis and inform the designer of excessive current densities and voltage drops.

Progress continues on the Network C (NC) simulation system. Major revisions have been made in both the MOS models and numerical procedures since the system was described in the last semiannual report. The system is now adding significant simulation capability to the Consortium toolset; our intent is to include it in a forthcoming release of our software.

The Consortium continues to distribute a CMOS/nMOS design system formed of tools developed at the UW as well as UC Berkeley, CMU, and MIT. This distribution benefits the VLSI community; it also provides the Consortium with useful feedback on the effectiveness of the tools.

2 Progress on Design Generators

(W. Winder, R. Nottrott, M. Liem, J. Baer, L. Snyder, L. McMurchie)

Based on a model for design generator construction (see Appendix A, "Declarative Descriptions ..."), a compiler has been implemented which analyzes the declarative description of a generator instance. The compiler and its supporting software have been implemented in a modular fashion to accommodate layout, schematic, mixed-mode and functional outputs. In the following discussion the programs analyzing the declarative descriptions and producing outputs are called backend programs. The programs which interact with the user of a generator to produce the declarative description of an instance are called frontend programs.

Currently there exist backend programs for both mixed-mode and schematic outputs. Both are based on leaf cells containing PostScript[1] commands The output files contain these leaf cells plus placement commands that orient the leaf cells according to the structure specified in the declarative description. When sent to a device capable to interpreting PostScript code, a diagram of the schematic or mixed mode representation will be produced. Such devices currently include a laser printer and a graphics terminal.

The process of translation of the generator model into an output by the backend program occurs in four steps: initialization and control, parsing, table building and semantic analysis. The backend software implemented so far includes approximately 7,500 lines of new source code plus 2,500 lines of code adapted from other tools.

Parsing is implemented via a function that creates a parse tree of the instance of the grammar described by the notation. The parsing function is created from the grammar described in a grammar file. This file contains the description of all legal notation instances for all generator outputs (schematic, layout, mixed and functional), and as such, will be used by all backend processes. The grammar file is inputed to a YACC to YACC translator[2], which adds YACC actions to the grammar that will cause the parse tree to be built.

A lexical analyzer accompanies the parsing function. The parsing function acts as described in the YACC manual, getting tokens from the lexical analyzer as needed, in an attempt to recognize grammar production rules. Details of operation are, as far as possible, left to the called function.

When parsing is complete and all tables have been built, analysis begins. In the schematic and mixed mode backends, analysis consists of drawing each leaf cell in the correct place on a virtual page. This process requires the evaluation of symbols. If the symbol is a leaf cell, a PostScript function is called to draw it. If the symbol is a composite, the geometric expression describing the symbol is decomposed. In this decomposition, evaluation of a symbol may cause recursive calls. At the end of the process, the virtual page is mapped onto the physical page.

The model for the decoder given in Appendix A has been implemented to the point that the

[1] PostScript is a graphics page description language. PostScript is a trademark of Adobe Systems Incorporated.
[2] YACC is a compiler generation program available on most UNIX systems.

decoder schematic and mixed mode diagrams can be generated. Similar work for the multiplier is in preparation.

The frontend programs that interact with the user of a generator to create the generator model from the user specifications are located in the requirements definition phase.

3 Energy Analysis of VLSI Circuits

(L. Snyder, A. Tyagi)

Due to engineering limitations on heat dissipation from a planar chip and a general trend towards energy conservation, energy efficiency of VLSI circuits has become an important issue in VLSI algorithm design.

Power dissipation has long been recognized as a limitation on nMOS designs. Despite the lower energy requirements of CMOS designs relative to nMOS, energy dissipation is an issue in CMOS as well. In the first place circuit designers tend to overdrive CMOS devices to obtain higher speeds. Additionally, in CMOS the power consumption is a direct function of the system frequency. Thus to be energy-conscious, care needs to be taken in the design of high performance CMOS architectures.

There are two phases to designing an energy efficient VLSI circuit. In the first phase one tries to understand the theoretical limits (complexity). This helps the designer set realistic goals for a design. For example, with energy complexity of at least n squared, an implementation linear in energy consumption would not be possible. In the second phase one tries to obtain a design as close as possible to the lower bound derived in the first phase. Of course none may exist that achieves that lower bound.

The paper in Appendix B is an attempt in the first direction. The paper analyzes the energy complexity of a very useful class of functions identified as normal transitive functions. A function is normal transitive if it embeds a computation of a shifting function. Examples include multiplication, convolution, shifting, and three-matrix products. The paper shows that when designing systolic architectures for these functions, it does not pay to worry about energy optimization, since the worst case and average case energy consumptions are equal. The trend towards integrating most of the architectural components onto fewer and fewer chips has encouraged the design of datapath and control components with systolic algorithms due to their simple communication structure. Thus, our result is useful inasmuch as it frees the designer from considering energy optimization as one of the objectives of the architectural design of this class of functions.

Another result of the paper reconfirms the intuitive belief that the designer can trade the speed of operation with the energy consumption. The paper establishes a lower bound on the worst-case switching energy for the class of functions called one-switchable. A function is one-switchable if there exist two input assignments differing in only one bit position, such that, going from one input assignment to another switches most of the output bits. Thus transitive

functions are a subset of one-switchable functions. The paper gives a quantitative tradeoff between speed and energy consumption for such functions.

Once the inherent limits on the energy consumption of a function are understood, one strives to find the most energy-efficient VLSI circuit implementation for it. The paper in Appendix C analyzes the energy complexity of a very commonly used VLSI structure, the PLA (This is a preprint of a paper to be presented at the Stanford VLSI Conference in March, 1987).

In this paper a comparison is made between the energy complexities of the static and dynamic design styles for a PLA. We show that the average energy consumption of a dynamic PLA exceeds that of a static PLA. We also show that, on average, a dynamic PLA is faster than a static PLA. In order to prove these results, we deal with a very general class of PLAs. If we are allowed to restrict this class, we can do more. In particular, we show that the choice of an optimal design style for a data path control PLA depends on the degree of parallelism of the data path. A high degree of parallelism may in fact favor a static designed PLA.

Our results give a mathematical justification (within the limitations of our model) for picking one design style over the other for a given application domain.

4 Hercules: A Power Estimator for MOS VLSI Circuits

(A. Tyagi)

Hercules is a stage-based MOS power estimator. The present CMOS version reports the average and peak load current due to the charging and discharging of capacitance in the curcuit. Hercules also reports the average and peak direct current due to both p and n channel devices being on during a slowly rising input signal. A tree-like data structure models the Vdd and ground distribution from the pins to sources/drains of devices. By combining the current requirements with the power distribution tree, checks can be performed to see if maximum current densities have been exceeded. Voltage drops from pins to sources and drains can also be reported.

The salient features of Hercules are as follows:

1. A linear average time algorithm employed for computing current levels is based on stage decomposition [Ous85] of a CMOS VLSI circuit. A stage is a chain of switches followed by either an output or a gate. The stages are traced out in a depth-first order enabling us to deal with the cross-coupled memory elements. The current implementation is an extension of a timing verification program, Crystal [Ous83]. Hence, it supports all the mechanisms of Crystal for flow specification.

2. An accurate switch-level model of short-circuit current in CMOS inverters and static circuits is derived. In CMOS circuits, slow input signal edges give rise to short-circuit current from Vdd to ground. The duration and magnitude of this current depend on the input signal slope, load and the device gain. All these factors can be encapsulated into

a single number, the rise time ratio, as observed by Ousterhout [Ous84] in a different context. The rise time ratio is the ratio of input signal slope to the native output signal speed. Most of the digital circuits are designed with only a fixed set of load and transistor sizes. The information about these structures can be extracted from SPICE runs on basic types of devices occuring in the circuit. Thus this model can predict the short-circuit current levels to within 20% of SPICE calculations. Ousterhout [Ous84] was the first to use this fact to model the effective resistances of devices in Crystal.

3. The metal tree idea of Wilson [Wil85] is extended to accomodate multilayer metal and loops. Typically, a VLSI circuit does not have many loops in its power distribution metal bus structure. We found that the comb structure, a very compact biconnected component, is the most common form of loop encountered in power buses. We are able to deal with them very efficiently using a depth-first search technique.

[Ous83] J.K. Ousterhout. Crystal: A Timing Analyzer for nMOS VLSI Circuits. In *Proceedings of 3rd Caltech Consference on VLSI*, Computer Science Press, 1983.

[Ous84] J.K. Ousterhout. Switch-Level Delay Models for Digital MOS VLSI. In *Proceedings of 21st Design Automation Conference*, ACM-IEEE, 1984.

[Ous85] J.K. Ousterhout. A Switch-Level Timing Verifier for Digital MOS VLSI. *IEEE Transactions on Computer Aided Design*, July 1985.

[Wil85] J. Wilson. Analysis of Power Requirements inside of nMOS Integrated Circuits. M.S. Thesis, Computer Science Dept., Oregon Graduate Center, Beaverton, 1985.

5 The Pyramid Machine

(S. Tanimoto, T. Ligocki, R. Ling)

Rapid analysis of images is a requirement in several areas of machine vision, including industrial parts inspection, visual navigation systems for robots, and medical diagnosis from real-time imaging devices. While parallel computers exist which are effective in solving some problems (e.g. large numerical problems), their interconnection networks are not very suitable for image processing, since much time is spent routing data from one processor to another. Vision applications would usually be much better served by parallel computers with image-oriented interconnection networks.

It is commonly suggested that in order to construct such an architecture, one may assign a processor to each cell (pixel) of the image and allow the processors to communicate directly with their immediate neighbors in the array. While such a distribution of processors is an important improvement over one processor or a row of processors, it still does not provide much support for the computation of global (non-local) characteristics of an image. A hierarchical structure

has been proposed as having most of the advantages of the processor-per-cell arrangement, while also possessing a capability for global computations.

We have proposed and built a hierarchical image-processing architecture that we call a pyramid machine[3]. This machine, being simultaneously parallel and serial, allows gradual formation of more and more global descriptions of image data in parallel. One advantage of this system is the multi-resolution data that is implicitly available. Another advantage is the short ($logN$) data paths from the pixels to the root of the pyramid. Yet another advantage is that such operations as median filtering can be efficiently performed.

The prototype pyramid machine recently completed has four levels with 64 processing elements (PE's) at the base level. A custom VLSI circuit called the "HCL chip" provides a 4x4 array of PEs and is used as the basic building block.

The HCL chip is intended to implement a piece of a pyramidal architecture in such a manner that it can be used in the building of a system of arbitrary size. In order to achieve this goal, the design required adequate functionality as well as processor density. It seemed clear that a full custom implementation was the only solution meeting these requirements. The HCL chip was designed in 4 micron nMOS using the Consortium design system and fabricated by MOSIS.

6 VLSI Tools Release

Release 3.0 of the Consortium design system has now been distributed to 135 sites. A new release of software is currently being prepared. It will contain the '86 Berkeley tools (including Magic) as well as a number of tools developed at the UW. These tools will include a number of design generators developed under this project as well as Coordinate Free LAP, the layout language in which these generators have been written. Also included will be the functional simulation system NC described in section 8.

The new release will be available for a tape charge to Universities, government contractors and industries affiliated with the Consortium.

7 Educational Offerings

For the fourth year in a row the Consortium offered an intensive class in digital CMOS design. The course is intended to provide industry engineers with the fundamentals of design and instruction in the use of CAD software. We view the course as fundamental to technology transfer between industry and the University research efforts in VLSI design.

[3] "A Prototype Pyramid Machine for Hierarchical Cellular Logic", by S. Tanimoto, T. Ligocki, and R. Ling (to be published).

The course is divided into two parts. During the first three days of lectures and labs participants learn the basic characteristics of MOS devices and use of simulation tools. A second sequence of lectures and labs covers layout and verification methods.

Twelve students from six local firms and three universities attended the most recent course. Three students elected to do a chip design which will be fabricated in 3 micron CMOS through MOSIS.

A seminar on fault simulation has been organized and will be held on December 18. Prominent researchers from both universities and industries have been invited to talk about recent developments as well as the application of fault simulators to industry problems.

8 Progress on the Network C Simulation System

(W. Beckett)

There have been three areas of development activity for Network C (NC) over the past six months.

1. THE MOS BEHAVIOR CALCULATION.

The MOS behavior calculation which was presented in the last semiannual report has been reimplemented using a different approach. The previous method used numerical integration to compute the response of subcircuits within a MOS system to input stimuli. This process was working reasonably well for circuits with rather large node capacitances (like 2 micron NMOS) but did not seem to work for circuits with small node capacitances (like 1.25 micron CMOS). The problem was that in order to make the integration stable for the small capacitance values, a small time step had to be used. This small time step, in turn, increased the execution time beyond practical limits.

A second problem was that the integration based method could not handle the case of zero node capacitance. Zero node capacitance does not come up in practice but it is still a useful modeling abstraction in some cases and it was felt desirable that NC models evaluate this case correctly.

It was decided to replace the numerical integration with a general non-linear equation solver based on Newton's method. While this approach involves substantially more computation per time point than numerical integration, it was believed that an order of magnitude reduction in the number of time points required could be achieved while maintaining an acceptable level of accuracy.

Practical implementations of Newton's method usually include some kind of line search along the Newton direction. This improves both the stability and the performance of the method. The typical algorithm computes the Newton direction, which requires solving a linear system (whose coefficient matrix is the Jacobian of the system) for the Newton correction and then

moving back along this correction until the norm of the residual vector or right hand side of the system is reduced. At this point, the Jacobian is recomputed and the process is repeated.

It was considered extremely desirable to avoid this recalculation of the Jacobian in as many cases as possible since each Jacobian approximation requires n function evaluations where n is the rank of the system of equations. Consequently the above method based on simple line searching was replaced with Broyden's 1965 Method. This method is described in Nonlinear Programming: Analysis and Methods by Mordecai Avriel, on page 354.

Broyden's method uses line searching in a similar manner to that described above but when it is time to obtain a new Jacobian, Broyden's method computes an approximation to the new Jacobian from the current Jacobian using the results of function evaluations already computed during the line search. In other words, you get the new Jacobian without any additional function evaluations.

Actually, Broyden's method is even better than indicated above because the updating formula can be transposed so that it will update the inverse Jacobian directly rather than the Jacobian. Using the formula in this form means that each successive Newton correction is produced using a simple matrix multiply rather than by solving a linear system. This is another substantial time-saver.

The drawback to Broyden's method is that it requires a good first approximation to the inverse Jacobian at the solution in order that the series of approximations to the inverse Jacobian converge. In running several tests of the method on general non-linear systems I have seen a number of practical cases in which the method can be very slow to converge. Fortunately, in the case of MOS circuits, the branch relationships are all produced by the same circuit element, namely the MOS transistor, and the model for this transistor is very nearly tri-linear with very smooth transitions between the regions. For networks consisting of such well behaved and gentle circuit elements, Broyden's method seems to work extremely well.

The following tests were performed using twenty-six transistor master-slave CMOS latch.

	Number of time points in forecast interval	CPU time (CYBER 170/750)
New Method (Newton-Broyden)	400	17.315
	200	8.858
	100	4.195
	40	1.799
Old Method Numerical Integration	400	4.165

If the same number of points are computed for each forecast, the new method is, as expected, substantially slower than the previous method, in this case about a factor of four. However, in the case of the integration method, fewer than 400 points produce an unacceptably inaccurate solution. An acceptable plot was produced using the new method at forty time points per forecast and in this case the run time is substantially less than the integration method, better than a factor of two.

2. NEW FOUR PARAMETER TRANSISTOR MODEL.

The simple DC MOS law transistor model previously used by NC has been replaced. The new model is based on the one described in the Design and Analysis of VLSI Circuits by Glasser and Dobberpuhl, page 95. For PMOS the NC formulation of this model is given by:

If

```
PHIFN   = K*T*LOG(ND/NI)/Q
GAMMA   = SQRT(2.0*ESI*Q*ND)/COX
T1      = SQRT(VBS + 2.0*PHIFN)
VTE     = VTO - GAMMA*(T1 - SQRT(2.0*PHIFN))
DELTA   = GAMMA/(2.0*T1)
VDSAT   = (VGS - VTE)/(1 + DELTA)
KS      = MU*COX(W/L)/2
```

then

Cutoff: VDSAT >= 0

IDS = -1.0E-12*VDS

Linear: VDSAT <= VDS

IDS = -KS*(2*VDSAT - VDS)*(1 + DELTA)*VDS

Saturation: VDSAT > VDS

ISAT = -KS*VDSAT**2*(1 + DELTA)

This model includes a second order drain current correction, DELTA, and the body effect. Also, a slight off resistance has been added.

Finally, the following empirical correction for channel length modulation in saturation found on page 110 of Glasser and Doberphul is also included.

IDS′ = IDS*(1 + (VDS - VDSAT)/(VA + VDSAT))

where VA = EA*L*SQRT(ND/NT) and EA = -5.0.

The presence of the off resistance and channel length modulation mean that this transistor has no region in which its current is independent of the source drain voltage. This improves the stability of the numerical method since it helps to avoid singular Jacobians.

3. CONTINUOUS TIME SYSTEM ANALYSIS AND EVALUATION.

A new facility has been developed for NC that allows the system to represent and evaluate models of analog circuits. The new facility includes support for procedural models of analog circuit elements like bipolar transistors, resistors, capacitors, diodes, and opamps. It also allows instances of these elements to be specified in the elements lists of circuit definitions. The syntax is the same as that used for all other NC circuit elements. Within the model bodies, the declarator

network float analog x;

is used to indicate the terminal of an analog device.

A new circuit analysis routine which is analogous to the MOS circuit analysis routine, collects closely coupled subnetworks of analog devices together into analog subnetworks. These subnetworks are evaluated at equally spaced time points by a general non-linear equation solver. This adds a continuous time computation capability to the main loop of NC which previously was purely based on discrete event scheduling.

The outputs from the analog subcircuits may drive other types of subcircuits. In turn, the inputs to analog subcircuits may be taken from other types of subcircuits. It is possible, for example, to couple the output of a functional level model directly to the input of an analog circuit level model. The outputs of the functional model are interpreted as ideal voltage sources in the analog model in this case. It is also possible to connect things the other way, in which case, the output node of the analog circuit would appear to the functional model sensing it as a periodic stream of events.

The non-linear equation solver used for analog circuit evaluation uses a technique I call residual sweeping which is described in a paper by H. T. Kung, "The Complexity of Obtaining Starting Points for Solving Operator Equations by Newton's Method", which appeared in Analytic Computational Complexity, J. Traub, Academic Press, 1976. The idea is that if X is not a solution to $F(X) = 0$, then $F(X) = R$, R being the residual. This being the case, X is a solution to the system $F(X) - R = 0$. The family of systems $F(X) - k * R = 0$ for k in $[0, 1]$ then contains the system for which a solution is sought but not known ($k = 0$) and a system for which the solution is known but not sought ($k = 1$). The object is to try to move X and k at the same time until X satisfies F and $k = 0$. NC sweeps k from 1 to 0 using a Fibonacci search.

The importance of this procedure is that when coupling functional and analog models together it is possible that the output of the functional model may change abruptly. This will mean that

the last available solution for the analog system, namely the last time point computed, will not be a good first guess for the computation of the current point since conditions have changed too drastically. (Note that these changes appear on input nodes which do not have corresponding terms in the Jacobian). In this kind of situation a conventional Newton implementation is quite likely to fail to conve. e. The residual sweep is a technique for getting a better first guess for the current point than just the last point. Indications so far are that the technique works very well.

With the inclusion of the analog circuit modeling capability, NC achieves closure over the complete range of models of lumped element electronic circuits and systems.

Declarative Descriptions for VLSI Generators†

Meei-Chineh Liem, Jean-Loup Baer,
Lawrence Snyder and Larry McMurchie

Department of Computer Science
University of Washington
Seattle, WA 98195

† Submitted to the Design Automation Conference, June 28 - July 1, 1987.

Research was supported in part by DARPA under contract MDA 903-85-K-0072.

Declarative Descriptions for VLSI Generators

Abstract

High-level descriptions which can precisely describe a circuit among multiple equivalent representations are introduced. Syntax and semantics for layout, mixed mode, schematics and functional descriptions are presented. They are illustrated by two examples: a decoder and a multiplier. These descriptions can facilitate the VLSI design process and serve as a comprehensive documentation tool.

1. INTRODUCTION

Design descriptions of VLSI components and systems are an "integral part of the design process" [Waxman 86]. In the context of the VLSI generator project in progress at the University of Washington, we present a declarative generator model which can be used to guide the generation process of a circuit. The description in the model is robust, natural, simple, expressive, and is structured in a hierarchical manner. It can precisely and abstractly describe a circuit across its multiple representations. It also serves as a comprehensive tool that documents the designer's ideas as well as the complexities of the circuit.

1.1. Design Generator Model

A *design generator* is defined as *a program that produces a family of circuit designs, each one solving a different instance of a particular problem. The input is a problem-specific set of parameters; an example for the output is a CIF (CalTech Intermediate Form) definition of the layout of the mask layers.* One of the objectives of the design generator research is to develop a generator construction methodology with appropriate abstractions, procedures and tools to assure production of correct, quality parts [UW/NW 84].

We use the term "**model**" to refer to a complex data structure that guides the generation process. The model fills the gap between the "high level" input and various outputs such as the layout. Figure 1-1 illustrates the role that the model plays in the generation process. The model is an overall static description of one instance of a circuit. It consists of leaf cells, a set of descriptions, and a catalog which includes the appropriate characteristics of this circuit. These components are produced by execution of the circuit specific generator software routines. Under the guidance of the model, the circuit-independent generator routines can create outputs such as the layout or the schematic diagrams at the gate or the transistor level of the circuit.

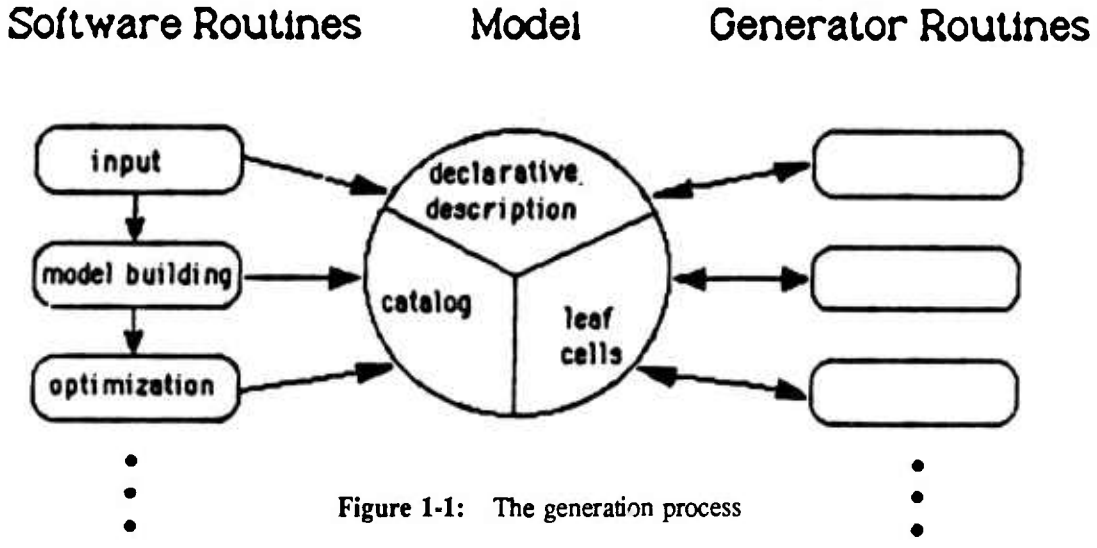

Figure 1-1: The generation process

The model should be sufficiently complete, so that a program which is guided by the model can generate the layout or other circuit descriptions. On the other hand, it should be sufficiently abstract to be able to capture the complex data structure of a design. This paper concentrates on the declarative description of the model.

1.2. Multiple Representation Problem

High level descriptions which can precisely describe a circuit across its multiple equivalent representations are needed in the design process. With abstraction, the descriptions can capture the information of how an instance of the circuit is built and how the circuits vary with the parameters. The declarative descriptions in the model will serve two functions: (1) design guide, and (2) documentation.

The declarative descriptions are a hardware description language(HDL). According to German & Lieberherr [German 85], HDLs can be divided into three categories:

The first category consists of languages that are purely functional specifications and do not necessarily imply a specific structure of the described circuits. One such language is μ FP (a variation of the Functional Programming language FP) [Sheeran 83] that describes both the semantics (behavior) of a circuit and its layout (a floor-plan).

In the second category we can place languages that allow both functional and structural specifications. They can be further divided into procedural and non-procedural classes. The latter offer safer descriptions than the former in the sense that more compile-time checks can be done. Among the procedural languages, we can cite ALI [Lipton 82] that is used to specify layouts free of design rule violations and Zeus [German 85] whose principles of structuring and much of the syntax are modeled after MODULA-2. Prolog [Suzuki 85], a non-procedural language, has been used for describing VSLI chips as concurrent building blocks connected by wires.

In the last category, we find languages that are only concerned with structure such as Regular Structure Generator (RSG) [Bamji 85] which uses previously defined cells to hierarchically build larger cells via *macro abstraction*, Escher [Clarke 85], a geometrical layout system for recursively defined circuits, and SLL, a Symbolic Layout Language [Ellis 81], that is the human-readable form of the schematic, logic, layout, and simulation information about a circuit in the Ruby database system used at the University of California, Berkeley.

2. DECLARATIVE DESCRIPTIONS

2.1. Overview

The high level descriptions which will be used for design and documentation in the generator project should have the following fundamental properties: (1) *Simplicity* and *Naturalness*, (2) *Expressiveness*, (3) *Abstraction* and *Hierarchical Structure*, and (4) *Technology Independence*. Each description describes an instance of a family of circuit designs in one of four possible representations: layout, schematic diagram (gate level and transistor level), and functional description. A description consists of two parts: (1) the *declarative part*, which includes the name of a circuit, the type of the representation, a list of parameters, a collection of leaf cells, and a set of imported functions; and (2) the *imperative part*, which consists of a collection of statements used to describe an instance of a circuit, e.g. a decoder with three select wires in NAND gate style. The description for the layout or the schematic representation can be regarded as a collection of *objects* (leaf cells or abstract objects) and a set of *relations* among these objects. For the functional description, the intermediate hidden mechanisms between inputs and outputs are described.

The syntax of this new high level description is designed to be as close to that of the "C" programming language as possible since the generators are written in C. The Extended Backus Naur Formalism (EBNF) definition for the declarative description can be found in [Liem 86].

2.2. Declaration

The syntax of the declarative part of a description is of the form:

NAME <circuit_name> ;

TYPE <representation_type> ;

PARAMETER <parameter_list> ;

LEAF CELLS <cell_list> ;

FUNC <function_list> ;

Boldface characters are used to indicate keywords and required syntactic elements. **FUNC** <function_list> is optional and **LEAF CELLS** <cell_list> are not used in the functional description.

The <representation_type> is either **LAYOUT, MIXED, SCHEMATIC** or **FUNCTIONAL**. <parameter_list> is a list of *inputs* to the circuit. They are arbitrarily chosen; however, the values that they are bound to in the declarative part are constant through the entire description. <cell_list> contains all the leaf cells

that are used in a description to generate a geometric representation (the layout or schematic diagram in gate level or transistor level) of a circuit. *Leaf cell* is the lowest level module in the hierarchy of a description and is part of the system library. Following the <cell_list>, the functions that aid in the circuit description are specified. Functions are user-defined. For example, *binary* can be a function which will return a binary representation of a number.

2.3. Objects

Leaf cells are the *primitive objects* to which geometric operators can be applied, and out of which more complex objects (*abstract objects*) can be built. In general, they have some predefined, sufficiently general, functionality. For example, it can be a NAND gate used for the schematic description of a decoder or a physical layout of a half-adder used for the layout description of a multiplier. A leaf cell can be *instantiated* as many times as desired by specifying the number of repetitions. For example, if **lcell** is a leaf cell, then | (**lcell** (n)) means to create an object which is a collection of n copies of **lcell** and the relations among these copies will be defined by the geometric operator |, i.e. vertical composition. A leaf cell without an argument is by default an instance of that cell in the user's working directory.

Abstract objects are created to provide designers with the mechanism to describe a circuit representation hierarchically so that most of the details at one level of the hierarchy are truly hidden from all higher levels. An abstract object can be defined recursively. An alias of a leaf cell, an array of leaf cells, a group of heterogeneous leaf cells, or a combination of the last two is an abstract object. Moreover, an array of abstract objects, a group of heterogeneous abstract objects or a combination of these two is also an abstract object. It should be noted that an abstract object represents an integrated consecutive part in the geometric placement. Generally, it is a module which has some functionality. Abstract objects can also be instantiated as many times as desired by providing the appropriate arguments. Thus, **--(row[i](i=0..4))** generates five objects whose relations are defined by the operator --, i.e. by horizontal composition. An element of an array of abstract objects can be accessed by specifying the subscript as in most programming languages. Abstract objects are considered global. Thus, object names within a description must be unique.

Given the features of leaf cells and abstract objects mentioned before, each description can use many levels of abstraction. At the highest level of the hierarchy, it is a single abstract object -- the name of the circuit that the designer intends to describe. At the lowest level of abstraction, the circuit is a collection of leaf cells. The description of a representation of a circuit is recursive in nature -- each abstract object is specified as a collection of lower level objects. Since an abstract object is defined after it is used, the description of an object promotes

"top down" design or "stepwise refinement".

2.4. Operators

The operators which are used in our declarative descriptions can be arranged in the following groups: (1) *geometric*, (2) *arithmetic*, (3) *relational*, (4) *logical*, and (5) *assignment* (=).

The geometric operators take objects as arguments and produce objects as results. The first four operators are used to combine objects into more complicated abstract objects. They are shown in Figure 2-1.

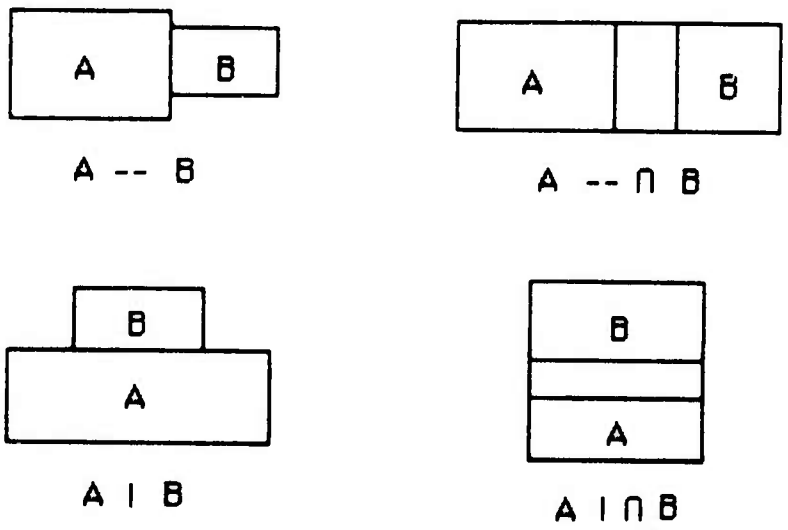

Figure 2-1: Geometric operators for combining objects

The last three operators are used for linear transformations. They are shown in Figure 2-2.

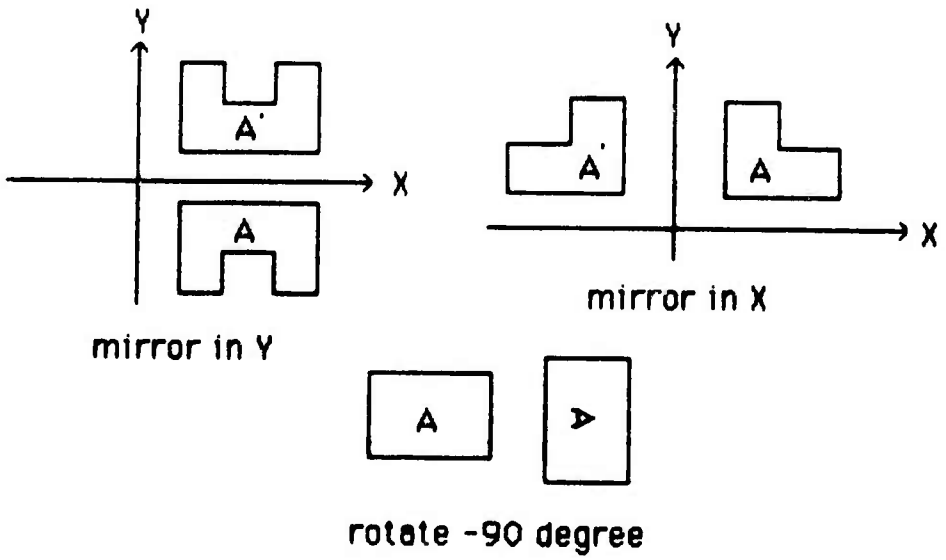

Figure 2-2: Geometric operators for linear transformations

All geometric operators have the same precedence level and they are collectively left-associative in the absence of parentheses. The *interface* (which will be implemented by using **registration marks**) between two objects is not explicitly specified since it is an implementation rather than a descriptive issue.

Geometric operators are only used in describing the layout or schematic representations of a circuit. For functional descriptions, arithmetic, relational and logical operators are used as in conventional programming languages. The arithmetic operators are +, -, *, /, ** (exponentiation), and % (modulus operator). The relational operators are <=, <, == (equal to), != (not equal to), >, and >=. There are two types of logical operators; the *logical connectives* include && (AND), and || (OR), while the *bitwise logical operators* include & (bitwise AND), | (bitwise inclusive OR), ^ (bitwise exclusive OR), and ~ (one's complement).

2.5. Flow of Control

Two fundamental flow-of-control constructs are provided to enhance the expressiveness of a description: **IF** (decision making) and **looping**. **IF** is used to specify conditions. Looping is expressed in the form of either (1) providing the number of times for repetition, for example, --(X(m)) means to create *m* horizontally joined instances of X, or (2) providing the upper bound, lower bound and step of the loop index, for example, |(X[i](i = 4 .. 0, -2) means to create 3 instances of X, with X[4] situated at the bottom, X[2] situated in the middle, and X[0] situated on the top.

3. MULTIPLE REPRESENTATIONS

A decoder[1] will be used throughout as a representative and simple example.

3.1. Layout Description

The layout description for an instance of a circuit describes how the leaf cells have to be displayed. The description is **hierarchical**, and uses *substitution*. That is, a bigger abstract object is composed of leaf cells or conceptually smaller abstract objects.

The following statements illustrate the layout description for a 3-to-8 NAND style decoder.

```
NAME   decoder;

TYPE   LAYOUT;
```

---

[1]The author of the decoder generator is Marty Sirkin.

```
PARAMETER   n = 3;

LEAF CELLS  dec_na_ll, dec_na_i_inv, dec_na_lc, dec_na_low,
              dec_na_high, dec_na_out, dec_na_one, dec_na_zero;

FUNC  binary;

MAIN

    decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));

    row [2**n] = dec_na_ll -- ( -- (dec_na_i_inv (n))) -- dec_na_lc;

    row [i] = dec_na_low -- select_wire [i]
            -- dec_na_high -- dec_na_out;

    select_wire [i] = ( -- (X [i,j] (j = n .. 1)));

    X [i,j] = dec_na_one, if binary (i,j) == 1
            = dec_na_zero, if binary (i,j) == 0.
```

The layout representation is shown in figure 3-1. The corresponding hierarchical structure is shown in Figure 3-2.

The fragment: `decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));`

creates an object named **decoder** which is made up of an abstract object, row [2^n], and 2^n *vertically* stacked abstract objects named row [2^n - 1], row [2^n - 2], ..., row [0]. row [2^n - 1] is situated on top of row [2^n], row [2^n - 2] on top of row [2^n - 1], etc. At this level of abstraction, row [2^n] and row [i] can be thought of as primitive components. Their lower level components are defined next.

The fragment: `row [2**n] = dec_na_ll -- (-- (dec_na_i_inv (n))) -- dec_na_lc;`

specifies the components of row [2^n] one level lower in the hierarchy. Row [2^n] consists of n horizontally joined instances of **dec_na_i_inv** with **dec_na_ll** on the left hand side and **dec_na_lc** on the right hand side. The elements in the arraylike structure **row** are further defined by:

```
row [i] = dec_na_low -- select_wire [i] -- dec_na_high -- dec_na_out;
```

For i with the values in the range 0 to 2^n - 1, row [i] is created by horizontally joining from left to right an instance of the leaf cell **dec_na_low**, an abstract object **select_wire [i]**, an instance of the leaf cell **dec_na_high**, and finally, an instance of **dec_na_out**.

Moving one level down in the description hierarchy, the abstract object **select_wire [i]** is described in terms of a collection of another abstract object, **X**. The fragment `select_wire [i] = (-- (X [i,j] (j = n .. 1)));` specifies that

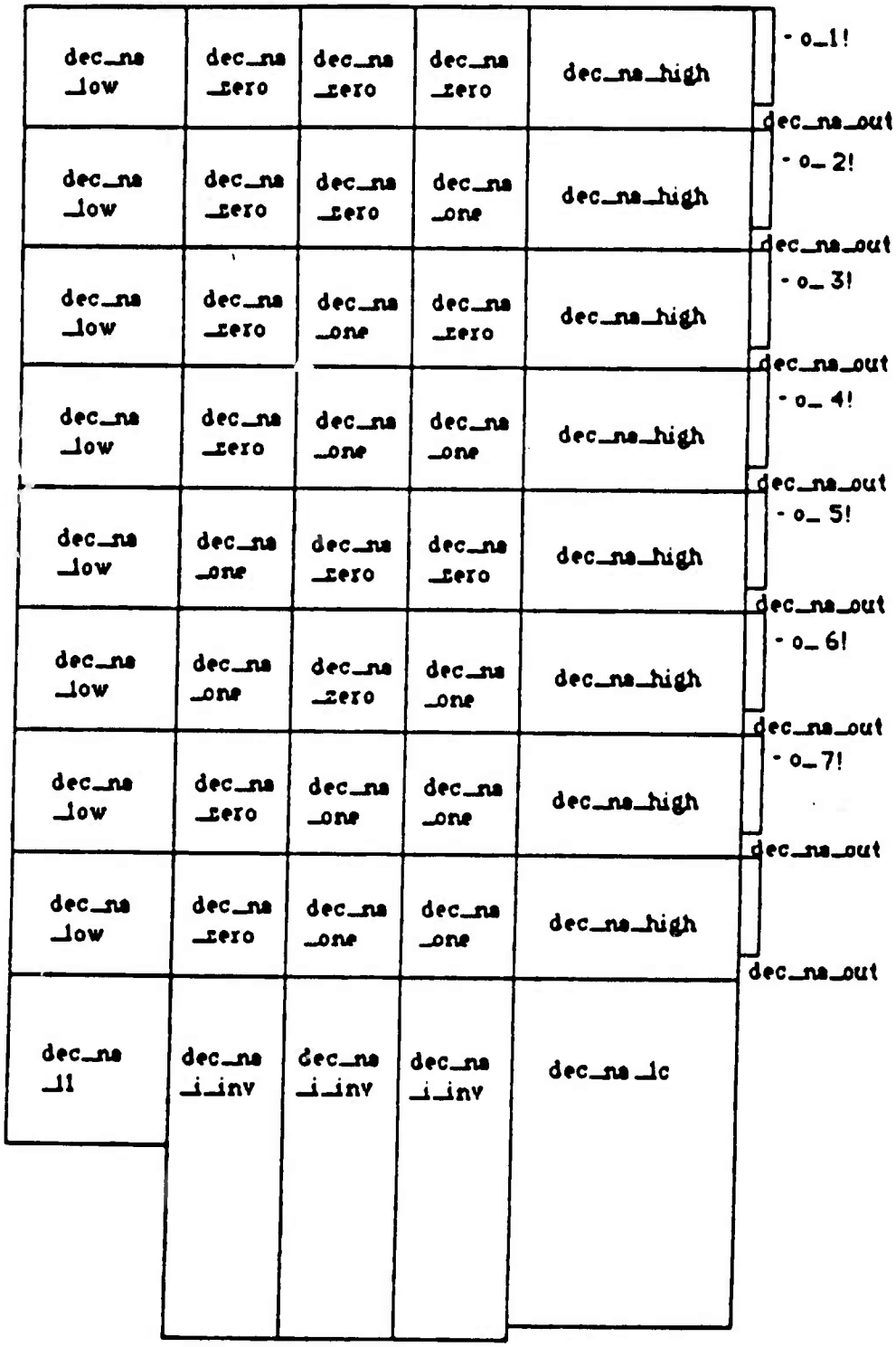

Figure 3-1: Layout representation

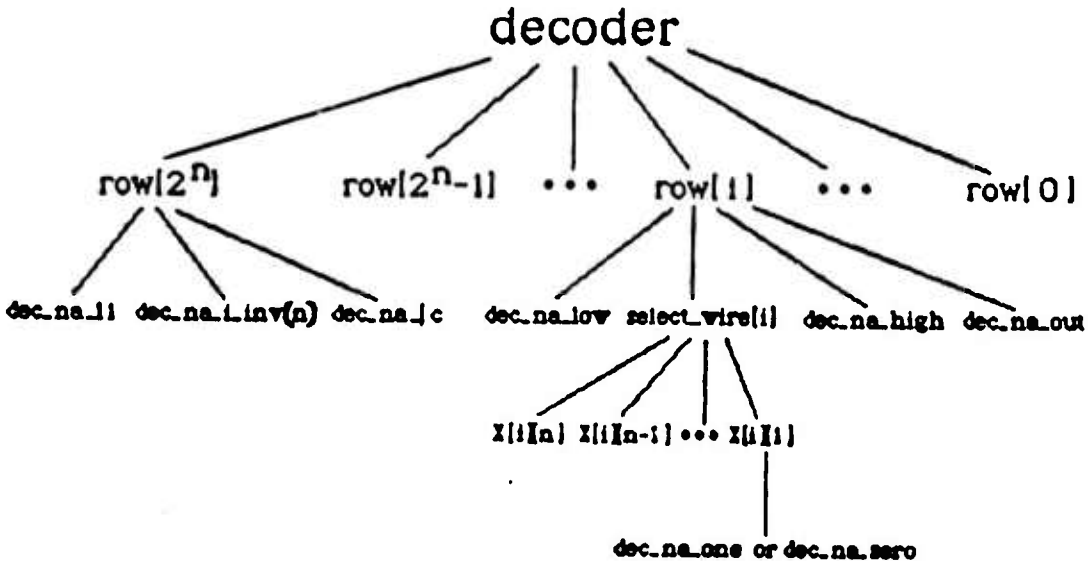

Figure 3-2: Hierarchy of objects in layout description

each **select_wire** consists of n horizontally joined instances of **X** with indices from n to 1, from left to right respectively. It is important to note that index i is required in the expression. It serves to distinguish different instances of **select_wire**. The definition of **X** clarifies this point. Abstract object **X** is defined as:

X [i,j] = dec_na_one, if *binary* (i,j) == 1
= dec_na_zero, if *binary* (i,j) == 0;

That is, if the *j*th bit of binary representation of *i* is 1, then substitute **dec_na_one** for **X**. Otherwise, substitute **dec_na_zero** for **X**.

Note that it is relatively easy to identify components of the circuit which are dependent on or independent of the parameter from the description. For example, the number of instances of **dec_na_i_inv** in row $[2^n]$ depends on the size of the input. However, the structure of row $[2^n]$ is invariant. In other words, **dec_na_ll** is always the leftmost component and **dec_na_lc** is always the rightmost component regardless of the number of instances of **dec_na_i_inv** in the middle. This observation is important in the understanding of a decoder over the entire space of parameters.

3.2. Mixed Mode Description

A mixed mode description illustrates how components (such as gates, transistors, and intersections of wires, etc.) are connected to perform a certain function. This representation can help create the logic network description for NETLIST and simulation. A mixed mode description is also hierarchical and uses substitution.

The mixed mode description for a 3-to-8 NAND style decoder is as follows.

```
NAME   decoder;

TYPE   MIXED;

PARAMETER   n = 3;

LEAF CELLS   gnd_mix, zero_mix, one_mix, in_mix, out_mix;

FUNC   binary;
.
MAIN

    decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));

    row [2**n] = (-- (in_mix (n)));

    row [i] = -- gnd_mix -- (-- (X [i,j] (j = n .. 1))) -- out_mix;

    X [i,j] = one_mix, if binary (i,j) == 1
            = zero_mix, if binary (i,j) == 0.
```

The leaf cells are shown in Figure 3-3.The leaf cells in the mixed mode description correspond to those in the layout description. For example, **zero_mix** and **dec_na_zero**, **one_mix** and **dec_na_one**, and **in_mix** and dec_na_i_inv perform the same functions. Figure 3-4 shows the expansion of the output of the mixed mode description (a schematic diagram at the transistor level) for a 3-to-8 NAND style decoder. The hierarchy of objects in mixed mode description is the same as in the layout case.

The close correspondence between these two descriptions is very important for design and documentation. The mixed mode representation provides one higher level of abstraction in circuit representation and is more immediately descriptive than the layout representation. Therefore, the corresponding part in the layout representation can become more understandable to the designer and users of the generator. Moreover, suppose that after verifying the electrical correctness of the circuit through the mixed mode representation, the designer decides to change part of the design. It is relatively easy to locate the corresponding part in the layout representation and make appropriate modifications.

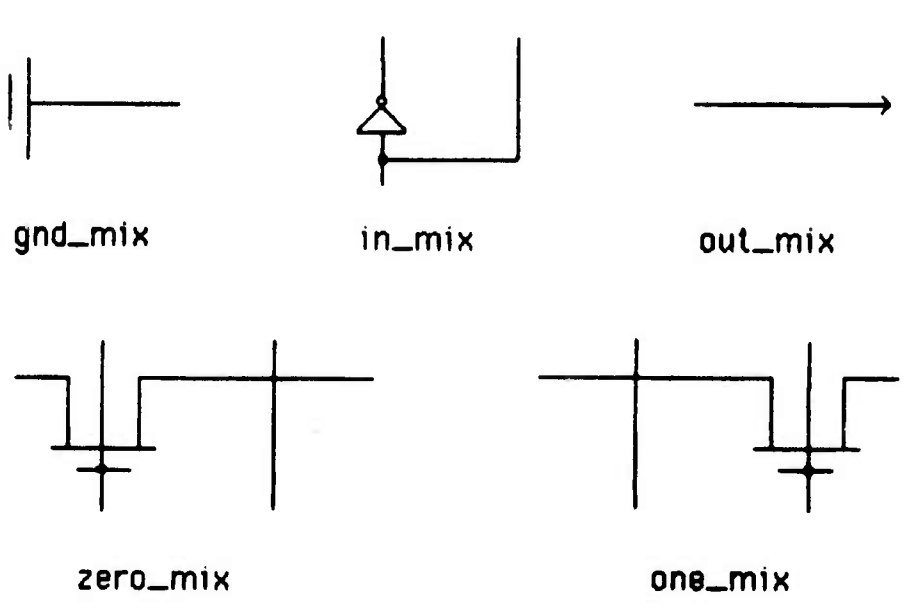

Figure 3-3: Leaf cells for mixed mode description

3.3. Schematic Description

The schematic description provides one higher, more descriptive level of abstraction than the mixed mode description in the sense that the graphical version of the former is at the *gate* level, while that of the latter is at the *transistor* level.

The following is the schematic description for a 3-to-8 NAND style decoder.

```
NAME   decoder;

TYPE   SCHEMATIC;

PARAMETER   n = 3;

LEAF CELLS   nand_sche, zero_sche, one_sche, no_connec_sche, in_sche;

FUNC   binary;

MAIN

    decoder = row [2**n] | ( | (row [i] (i = 2**n - 1 .. 0)));

    row [2**n] = (-- (in_sche (n)));

    row [i] = (-- (X [i,j] (j = n .. 1))) -- nand_sche;

    X [i,j] = (| (C [i,j,k] (k = 1 .. n)));
```

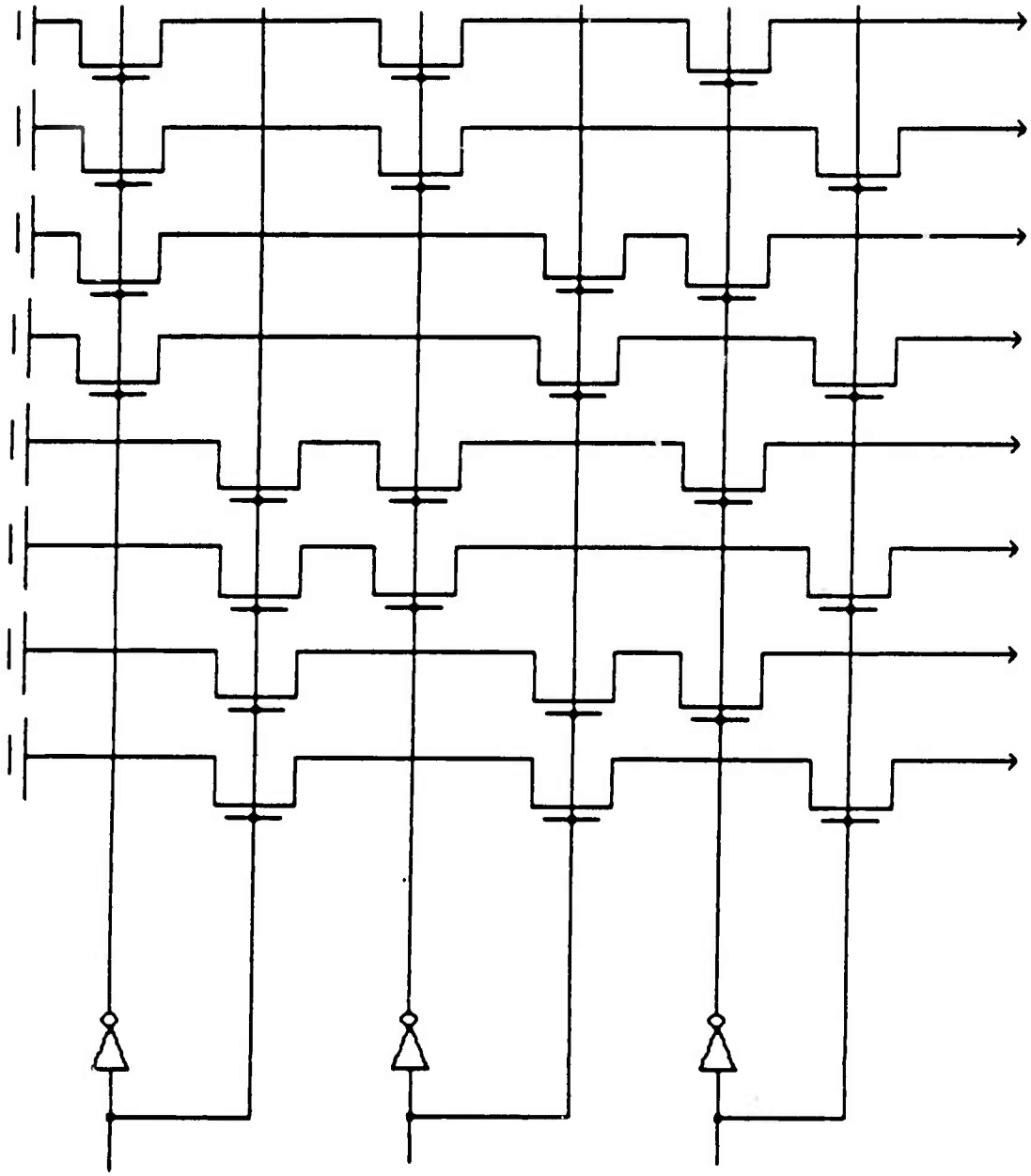

Figure 3-4: Mixed mode representation

C [i,j,k] = one_sche, if *binary* (i,j) == 1 && k == j
= zero_sche, if *binary* (i,j) == 0 && k == j
= no_connec_sche, if k != j.

Figure 3-5 shows the internal details of each leaf cell, and figure 3-6 shows the schematic diagram of a 3-to-8 NAND style decoder.

The hierarchical structure of the objects used in the schematic description is shown in Figure 3-7. It is

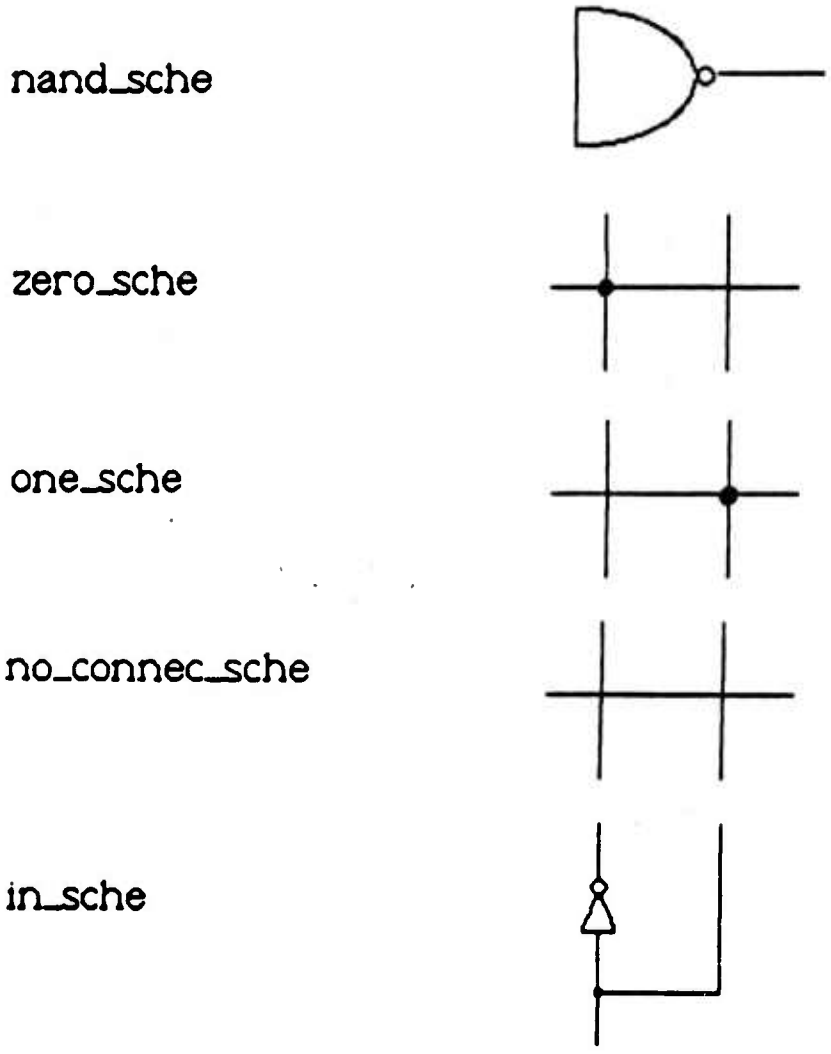

Figure 3-5: Leaf cells for schematic description

important to note that the hierarchy is extended one level lower, but the correspondence with other descriptions still holds.

3.4. Functional Description

The previous three descriptions specify the *geometric relations* of objects. In contrast, the functional description does not specify the relative positions of objects, but *how* a particular design should respond to a given set of inputs. In other words, the *algorithm* to be performed by a circuit is described. The following statements illustrate the functional description for a 3-to-8 NAND style decoder.

```
NAME   decoder;

TYPE   FUNCTIONAL;
```

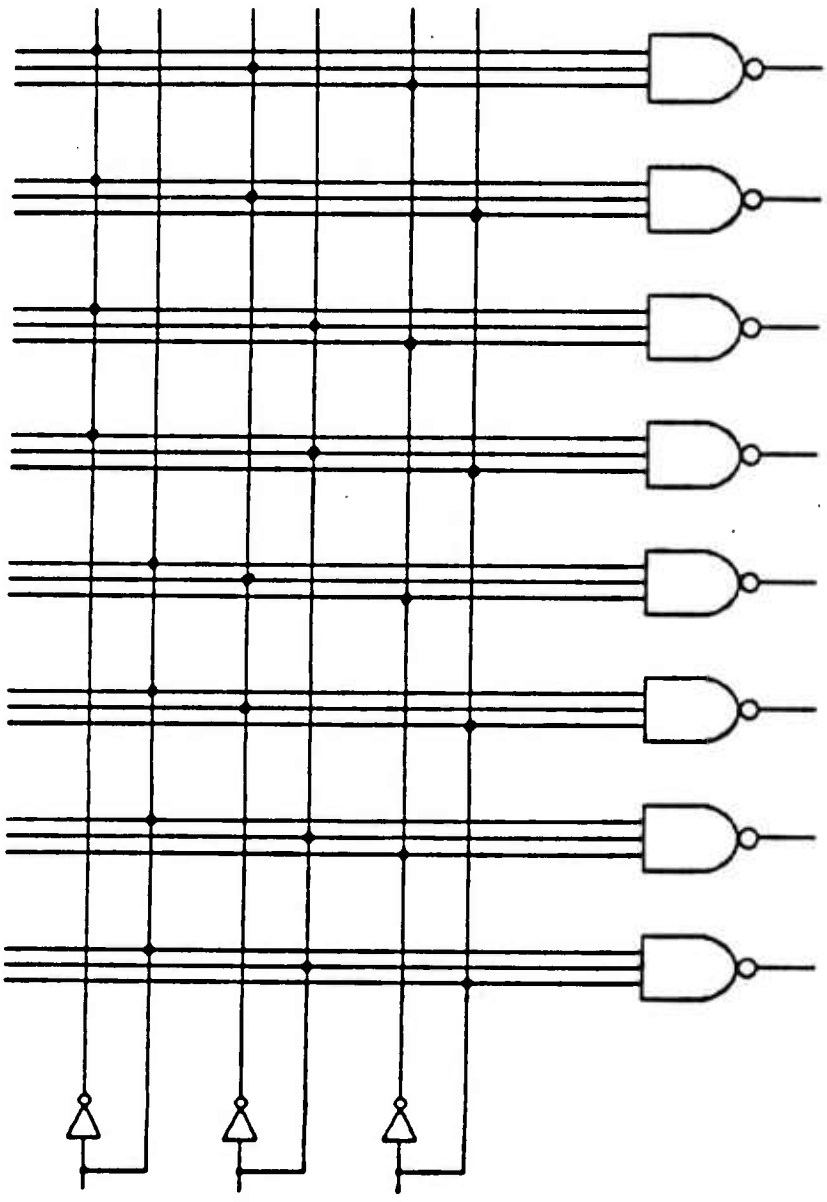

Figure 3-6: Schematic representation

```
PARAMETER   n = 3;

FUNC   nand, binary;

MAIN

    decoder = OUTPUT [i] (i = 0 .. 2 ** n - 1);

    INPUT = A [j] (j = n .. 1);

    OUTPUT [i] = nand (X [i,j], j = n .. 1): <timing_specification>;
```

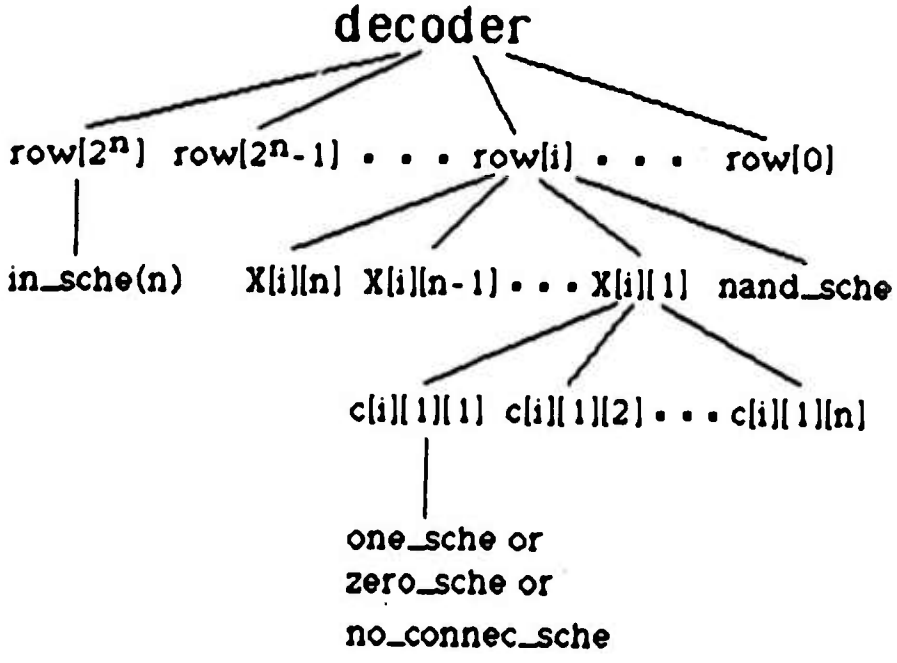

Figure 3-7: Hierarchy of objects in schematic description

X [i,j] = *binary* (i,j) * A[j] + ~ *binary* (i,j) * ~ A[j].

The imported function *nand* simulates the function of a NAND gate. The statement

decoder = OUTPUT [i] (i = 0 .. 2 ** n - 1);

denotes that there are 2^n outputs named OUTPUT [0], OUTPUT [1], ..., and OUTPUT [2^n - 1]. A similar notation applies to INPUT. Each input and output is one bit wide. These two statements correspond to the first two statements in the schematic description: OUTPUT [i] is functionally equivalent to row [i] and both row [2^n] and A [j] (j = n .. 1) deal with the inputs (the complement of A [j] is specified by the complement operator ~).

OUTPUT [i] is further defined by the function *nand*:

OUTPUT [i] = *nand* (X [i,j], j = n .. 1): *<timing_specification>*;

The *<timing_specification>* specifies when the output bit becomes stable and available for an external circuit. It can be expressed in terms of i, clock period (T), and time delay (td). The values of T and td are technology and implementation dependent.

Depending on the *j*th bit of the binary representation of *i*, i.e. *binary* (i,j), the value of X [i,j] is either the input A [j] or its complement $\overline{A[j]}$. Note that given an OUTPUT [i], the binary representation of i corresponds to

the inputs A[n] A[n-1] ... A[j] ... A[1], where A [n] * 2^{n-1} + A [n -1] * 2^{n-2} + ... + A [1] * 2^0 = i. Note also that for an asserted OUTPUT [i], the inputs to the NAND gate must be all 1's. The algorithm is thus the following:

X [i,j] = A [j], if *binary* (i,j) == 1;
X [i,j] = $\overline{A[j]}$, if *binary* (i,j) == 0;

Since *binary* (i,j) is either 1 or 0, the algorithm can be simplified by stating that

X [i,j] = *binary* (i,j) * A [j] + ¯ *binary* (i,j) * ¯ A [j].

While the functional description uses arithmetic operators rather than geometric operators to describe a circuit, it still corresponds with the other descriptions. As before, the mechanism which is used is substitution and the structure of the description is hierarchical. The major difference is that the leaves of the tree now are inputs or their complements while the leaves of the trees in other descriptions were leaf cells.

4. A MORE COMPLEX EXAMPLE -- MULTIPLIER

This section will present a more complex example, a **multiplier**, to illustrate the versatility of the notation. mult is a generator for constructing an M x N cmos multiplier layout[2]. A 3 x 3 signed two's complement multiplier is chosen as an example in our discussion. We only show the schematic and the functional descriptions here. The layout and the mixed mode descriptions can be found in [Liem 86].

4.1. The schematic Description

The schematic description for a 3 x 3 signed two's complement multiplier is as follows. Figure 4-1 shows the expansion of the schematic description.

```
NANE   multiplier;

TYPE   SCHEMATIC;

PARAMETER   m = 3, n = 3;

LEAF CELLS   SignExt, FullMult, LSignExt, Comp, RComp, Add;

MAIN

    multiplier = adder | row[n] | (| (row[i] (i = n - 1 .. 1)));

    row[i] = SignExt -- (-- (FullMult (m - 1)));
```

[2]The author of the multiplier generator is Wayne Winder.

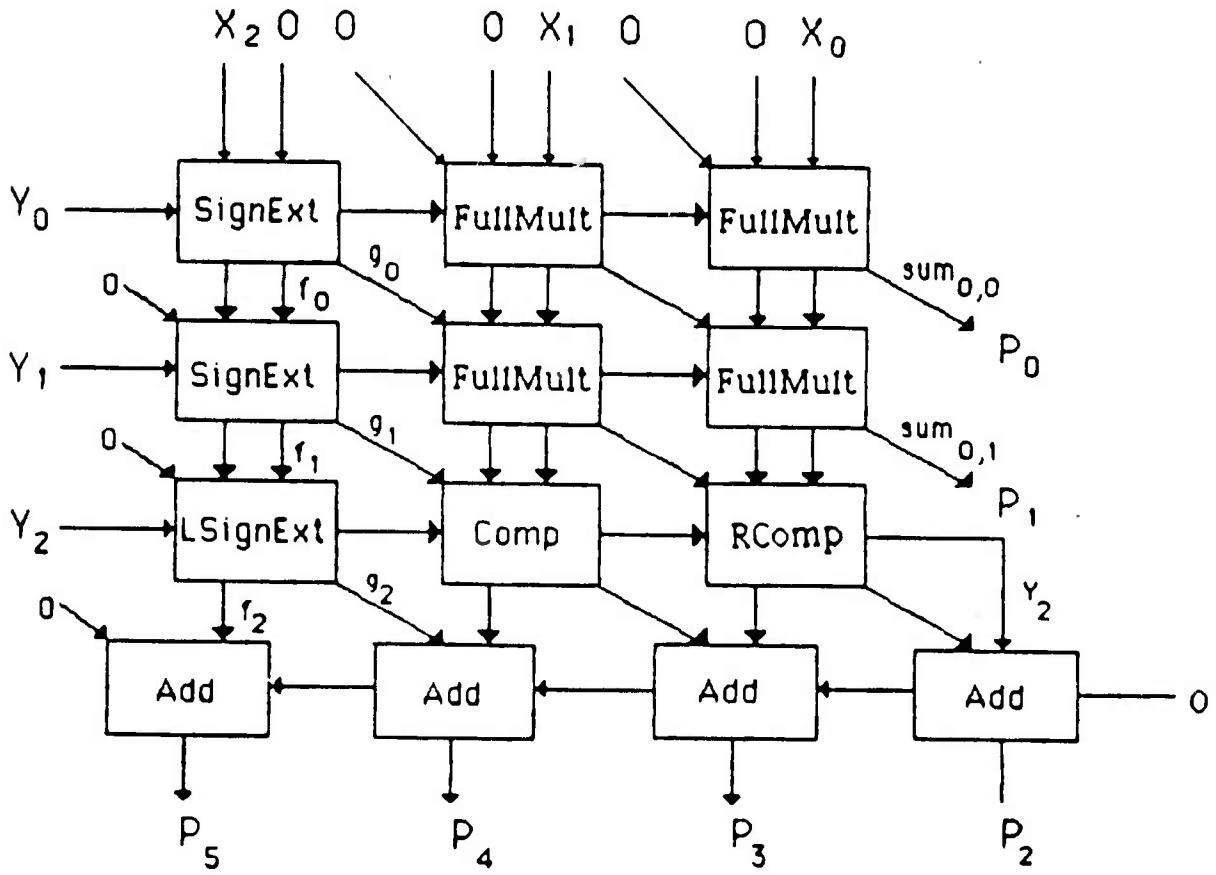

Figure 4-1: Schematic diagram of a 3 x 3 multiplier

row[n] = LSignExt -- (-- (Comp (m - 2))) -- RComp;

adder = (-- (Add (m + 1))).

To generate a signed two's complement multiplier, six leaf cells are needed. **SignExt** generates the sign extension bits f_j and g_j for $0 \le j \le n\text{-}2$, while **LSignExt** evaluates the last sign extension bits, f_{n-1} and g_{n-1}. The function of **FullMult** is to calculate the sum of $carry\_out_{i,j}$, $sum\_out_{i+1,j}$ and the ANDed function of X_i and Y_j for $0 \le i \le m\text{-}2$ and $0 \le j \le n\text{-}2$. **Comp** and **RComp** evaluate the sum of $carry\_out_{i,n-2}$, $sum\_out_{i+1,n-2}$ and the ANDed function of $\overline{X_i}$ and Y_{n-1}, $0 \le i \le m\text{-}2$. They are basically the same with the exception that Y_{n-1} in **RComp** exits from the right hand side and curves down to the first bit of the ripple adder. Y_{n-1} is one of the inputs to the lowest bit of the ripple adder because of the last term in the expanded version of the *n*th partial product, $Y_{n-1}2^{n-1}$. The leaf cell **Add** computes the sum of three inputs and produces one bit of the final product. In addition, the carry out is sent to the next higher order bit position of the ripple adder.

Thus row[i] accomplishes the process of forming the partial product, adding it with the previous cumulative sum and performing the sign extension. The function of row[n] is to perform the ANDing of the complement of multiplicand and Y_{n-1} as well as generate the cumulative sum and the last pair of sign extension bits, f_{n-1} and g_{n-1}. The object adder generates the higher order $m+1$ bits of the final product. The algorithm used for the multiplication is not evident from this description. Hence, we need a functional description.

4.2. The Functional Description

The functional description describes how the final product is generated given a multiplicand X with m bits and a multiplier Y with n bits. At the highest level, we could simply say that OUTPUT = X * Y. While this might be sufficient for some high-level functional simulation, further details will often be needed. The next level is described as:

```
NAME   multiplier;

TYPE   FUNCTIONAL;

PARAMETER   m = 3, n = 3;

MAIN

   multiplier = OUTPUT[l] (l = 0 .. m+n-1);

   INPUT = X[i] (i = 0 .. m-1),
           Y[j] (j = 0 .. n-1);

   OUTPUT = X * Y
          = CS[n];

   CS[k] = CS[k-1] + PP[k], if 1 ≤ k ≤ n;

   CS[0] = 0;

   PP[k] = Y[k-1] * X * 2**(k-1), if 1 ≤ k ≤ n;

   PP[0] = 0.
```

PP[k] represents the *k*th partial product and CS[k] represents the *k*th cumulative sum of the partial products. Note that the details of the extension of sign bit and the separation of carry and sum in each cumulative sum are not explicitly described here. This functional description corresponds to the given schematic description. The first n-1 rows of row[i] correspond to the first n-1 CS[k]'s while row[n] and the ripple adder are implicitly accounted for by evaluating CS[n].

At the lowest level of description, the generation of each bit of the product output and sign extension bit is shown.

```
NAME   multiplier;

TYPE   FUNCTIONAL;

PARAMETER   m = 3, n = 3;

FUNC   sum, carry, summation;

MAIN

   /*   TERMINOLOGY:
     CS[k] = cumulative sum of partial products; made up of partial
             sum and partial carry. 1 ≤ k ≤ n.
     PS[k] = partial sum; the sum portion of the result of an addition
             when the carry overs are not rippled through the higher
             order bits.
     PC[k] = partial carry; the carry portion of the result of an
             addition when the carry overs are not rippled through
             the higher bits.
             Note: PS[k] + PC[k] = CS[k] the total result of the
             addition.
     F[k]  = the higher order bit resulting from extending the sign
             bit during an addition. This is also the MSB of PC[k].
     G[k]  = the lower order bit resulting from extending the sign
             bit during an addition. This is the MSB of PS[k].
     RS[l] = sum bit generated by the ripple adder. This is also
             one of the product bits. n-1 ≤ l ≤ m+n-1.
     RC[l] = carry bit generated by the ripple adder.
     PP[k] = partial product; it is one bit of Y times the vector X,
             then shifted appropriately.
     Fn[k,l] = bit with the 2**l power of the kth evaluation
             of the function Fn.      */


   multiplier = OUTPUT[l] (l = 0 .. m+n-1);

   INPUT = X[i] (i = 0 .. m-1),
           Y[j] (j = 0 .. n-1);

   OUTPUT = X * Y
          = RippleSum;

   RippleSum = summation (RS[t]*2**t, t = m+n-1 .. n-1) +
               summation (OUTPUT[t]*2**t, t = n-2 .. 0);

   RS[l] = sum (PS[n,l], PC[n,l], RC[l]), if n-1 ≤ l ≤ n+m-2;

   RS[m+n-1] = sum (0, PC[n,m+n-1], RC[m+n-1]);
```

RC[*l*+1] = *carry* (PS[n,*l*], PC[n,*l*], RC[*l*]), if n-1 ≤ *l* ≤ n+m-2;

RC[n-1] = 0;

PC[n,n-1] = Y[n-1];

/* cumulative sum is made up of partial sum and partial carry */
/* PS[k] and PC[k] are not added until next addition */
CS[k] = PS[k] + PC[k], if 1 ≤ k ≤ n;

PS[k] = *sum* (PS[k-1], PC[k-1], PP[k])
= *summation* (PS[k,t]*2**t, t = m+k-2 ..k) +
summation (OUTPUT[t]*2**t, t = k-1 .. 0);

PS[0] = 0;

/* The MSB of a partial sum is G[k] */
PS[k,m+k-2] = G[k];

PC[k] = *carry* (PS[k-1], PC[k-1], PP[k])
= *summation* (PC[k,t]*2**t, t = m+k-1 .. k);

PC[0] = 0;

/* The MSB of partial carry is F[k] */
PC[k,m+k-1] = F[k];

F[k] = F[k-1] | PP[k-1,m+k-2], if 1 ≤ k ≤ n-1;

F[0] = 0;

G[k] = F[k-1] ^ PP[k-1,m+k-2], if 1 ≤ k ≤ n-1;

G[0] = 0;

F[n] = F[n-1] | (~X[m-1]*Y[n-1]);

G[n] = F[n-1] | (~X[m-1]*Y[n-1]);

PP[k] = Y[k-1]*X*2**(k-1)
= *summation* (PP[k,t]*2**t, t = m+k-2 .. k-1), if 1 ≤ k ≤ n-1;

PP[n] = (X[m-1]*Y[n-1] - Y[n-1])*2**(m+n-2) +
summation (Y[n-1]*~X[t]*2**(t+n-1), t = m-2 .. 0) + Y[n-1]*2**(n-1);

OUTPUT[*l*] = RS[*l*]: <*timing_spec*>, if n-1 ≤ *l* ≤ n+m-1
= PS[*l*+1,*l*]: <*timing_spec*>, if 0 ≤ *l* ≤ n-2.

Three functions, *sum*, *carry* and *summation*, are used. *Sum* (a,b,c) is equal to **a XOR b XOR c** and *Carry* (a,b,c) is equal to **ab OR bc OR ac**. *Summation* is the addition of terms in the series. This function simulates the expansion of $\sum_{t=low}^{high} X_t$. Thus, *summation* (RS[t]*2**t, t = m+n-1 .. n-1) represents

$RS[m+n-1]2^{m+n-1}+RS[m+n-2]2^{m+n-2}+...+RS[n-1]2^{n-1}$.

This functional description is very close to the actual implementation of the algorithm. The multiplication is a sequence of carry-save additions with one ripple addition at the end. The 2's complement notation implementation requirements are handled in the sign extension(f and g) and the generation of the last partial product PP[n].

5. CONCLUSIONS

In this paper, we have described the multiple representation problem and proposed a model which provides descriptions of the multiple equivalent representations of instances of a circuit for design and documentation purposes. A set of notations to be used in the various descriptions has been introduced. These notations have been created to make the descriptions simple, natural, expressive, and to show abstract, hierarchical structure and technology independence. Two examples, a decoder and a multiplier, have been used to illustrate the application of the descriptions.

For each design, there are some common characteristics among the four different views of description. They are

- The descriptions are declarative.
- The hierarchical decomposition of the design proceeds recursively. Multiple levels of abstraction make it possible to suppress unnecessary details and make the design more comprehensible.
- Substitution is the mechanism to navigate in the hierarchical description.
- There is a correspondence between different descriptions. As a result, the change of design across different descriptions can be made easily.

Various extensions to the declarative descriptions can be done to form a more powerful and flexible model that guides the generation process. Two areas of research which need further exploration are: *building a translation system which can generate the appropriate outputs for different descriptions of a circuit;* and *creating a design database which can organize the design data across the multiple representations of a design.*

References

[Bamji 85] Bamji, C., Hauck, C. and Allen, J.
A Design by Example Regular Structure Generator.
In *22nd Design Automation Conference*, pages 16-22. IEEE, 1985.

[Clarke 85] Clarke, E. and Feng, Y.
Escher--A Geometrical Layout System for Recursively Defined Circuits.
Research Report CMU-CS-85-150, Department of Computer Science, Carnegie-Mellon University, July, 1985.

[Ellis 81] S. Ellis.
A Symbolic Layout Language & Database for an Integrated VLSI Design System.
Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, December, 1981.

[German 85] German, S. and Lieberherr, K.
Zeus: A Language for Expressing Algorithms in Hardware.
IEEE Computer :55-65, February, 1985.

[Liem 86] Meei-chiueh Y. Liem.
Declarative Descriptions for VLSI Generators.
Master's thesis, Department of Computer Sciences, University of Washington, June, 1986.

[Lipton 82] Lipton, R., North, S., Sedgewick, R., Valdes, J., and Vijayan, G.
ALI: A Procedural Language to Describe VLSI Layouts.
In *19th Design Automation Conference*, pages 467-474. IEEE, 1982.

[Sheeran 83] Mary Sheeran.
μ FP - An Algebraic VLSI Design Language.
PhD thesis, Oxford University Computing Laboratory, November, 1983.

[Suzuki 85] Suzuki, N.
Concurrent Prolog as an Efficient VLSI Design Language.
IEEE Computer :33-40, February, 1985.

[UW/NW 84] UW/NW VLSI Consortium.
Quality VLSI Design Generators.
1984
A Research Proposal Submitted to The Defense Advanced Research Projects Agency Information Processing Technology Office by Department of Computer Science's University of Washington/Northwest VLSI Consortium.

[Waxman 86] Waxman, R.
Hardware Design Languages for Computer Design and Test.
Computer (4):90-97, April, 1986.

The Energy Complexity of Transitive Functions†

Lawrence Snyder & Akhilesh Tyagi*
Department of Computer Science
University of Washington
Seattle, WA 98195

†Proceedings of the 24th Allerton Conference on Control, Communications and Computing, Allerton Park, IL, Oct. 1-3 '86.

* Supported in part by DARPA under Contract MDA 903-85-K-0072.

The Energy Complexity of Transitive Functions [1]

LAWRENCE SNYDER & AKHILESH TYAGI
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

We define a *normal* transitive function to be a function that embeds a computation of the permutation group generated by the cycle $(1\ 2\ 3 \ldots n)$. Despite this restriction, the class of normal transitive functions is rich enough to include all the functions that Vuillemin showed to be transitive. A partial list consists of shifts, cyclic shifts, multiplication, convolution and linear transform. We show that for an implementation of these functions, where every wire is allowed to switch at most once, the average switching energy, $E_a(C)$, is $\Omega(n^2)$. If a wire is allowed to switch more than once then we prove a lower bound of $E_a(C) = \Omega(n^{3/2})$. We prove that *word systolic systems* for transitive functions consume the same order of energy both on average and in worst case. In particular, the average case energy $E_a(C)$ is $\Omega(A)$ and the worst case energy $E_w(C)$ is $O(A)$, where A is the area of any embedding for a circuit C to compute a transitive function. However, to demonstrate that this unexpected behavior is not universal, we show the existence of a systolic system for a problem for which the average case and worst case energy consumptions can be separated. We also extend a result by Kissin on a lower bound on the worst case switching energy of 1-switchable functions. For any convex embedding of a circuit to compute a 1-switchable function within depth $d(n)$, $log^2 n \leq d(n) \leq n^{\epsilon}$, with $0 < \epsilon \leq 1$, $E_w(C) \times d(n)$ is $\Omega(\max(n \log n,\ n\, d(n))$. Note that every transitive function is also 1-switchable, implying that these lower bounds apply to transitive functions, as well.

1 Introduction

Due to engineering limitations on heat dissipation from a planar chip and a general trend towards energy conservation, energy efficiency of VLSI circuits has become an important issue in VLSI algorithm design. Despite relatively low energy consumption of CMOS technology, the energy dissipation is still an issue in the design of high performance systems for the following reasons. The circuit designers tend to overdrive the CMOS devices for higher speeds. Worst still, in CMOS technology, the power consumption is a direct function of the system frequency. Thus high performance architectures need to be designed even more carefully, to be energy conscious. Most high performance systems are based on the concept of pipelining or systolic dataflow. The trend towards integrating most of the architectural components onto fewer and fewer chips has encouraged the design of datapath and control components with systolic algorithms due to their simple communication structure. Vuillemin [10] identified an important class of vfunctions known as *transitive* functions. Interestingly, this class encompasses all the data path functions like multiplication and shifting. Thus, our result has a wide impact in as much as it frees the designer from considering energy optimization as one of the objectives of the design. The second result reconfirms the widely held belief that the designer can trade the speed of operation with the energy consumption.

With the wider availability of VLSI design environments, and introduction of systematic design methodology by Mead and Conway [7], relatively inexperienced designers are transcribing their algorithms into silicon. Often, such engineering applications include multiplication, convolution and linear transforms. Interestingly, all these functions are normal transitive functions.

[1] Supported in part by DARPA under Contract MDA903-85-K-0072

Lengauer and Mehlhorn [6] have shown that for a function with $AP^2 = O(n^2)$, switching energy is bounded from below by $\Omega(AP)$, where A is the area and P is the period of a pipelined computation. Kissin [5] proved that for some monotonic circuits, switching can be superlinear, but switching energy is still bounded by $O(A)$. Kissin [4] explores switching complexity of *comparison, or* and *addition* functions. She shows that for 1-switchable functions, $\Omega(n \log n)$ worst case switching energy is required if computation is to be performed within depth $O(\log n)$. She also gives a linear average energy layout for an adder.

We first describe the VLSI model of computation and energy consumption. We prove a lower bound of $\Omega(n^2)$ on the average switching energy of normal transitive functions. In section 4, we show that the lower bound on average switching energy of a transitive systolic system matches the upper bound on the worst case switching energy. We conclude with the result about depth switching-energy trade-off. For a detailed version of this work, the reader is referred to the technical report [8].

1 Model

We will formally define the model in which we charge for energy. Our energy model is the same as the one originally outlined by Kissin [5]. For the sake of completeness, we will briefly describe the model. The VLSI model is the commonly accepted one, proposed by Thompson [9]. A layout can be viewed as an embedding of the communication graph in a rectesian grid. Each grid point can either have a processor or a wire passing through. A wire can not go through a grid point unless it is a terminal of the processor at that grid point. Wires have unit width and bandwidth and processors have unit area. The initial data values are localized to some constant area, to preclude an encoding of the results.

Let u be the unit switching energy defined to be the energy spent when a wire of unit length switches (changes state either from a $1 \rightarrow 0$ or from a $0 \rightarrow 1$). Note that a minimum size processor also consumes $O(u)$ energy when it changes state. We will say that a wire of length l consumes $\Omega(l)$ energy when it switches. It is implicit here that u is a technology dependent constant.

We do not account for the switching energy consumed by the processors. This is in accordance with the widely held belief that the wires take up most of the area in a VLSI layout. Since we are working with the lower bounds on switching energy, energy consumption of the wires definitely provides a lower bound on the total energy consumed.

We work with the Uniswitch Model (USM), as defined in Kissin [5]. In this model, a signal can propagate along a wire of arbitrary length in constant time. This restricts every wire in an acyclic circuit to switch at most once, for unpipelined computation. A symmetric notion is that of the Multiswitch Model. Kissin shows some upper bounds on switching energy of certain circuits in the Multiswitch model.

ASSUMPTIONS:

1. Each node in the circuit depends on an input, *i.e.*, for each node, a pair of inputs exists which makes that node switch.

2. All the inputs are synchronous. In other words, they are applied at the input ports, simultaneously.

3. Circuits are synchronous.

We denote a circuit $C(V,W)$ as a graph, where V is the set of nodes and W is the set of edges or wires in $V \times V$.

Definition 1 *A circuit $C(V,W)$ is said to be in state $s : V \cup W \longrightarrow \{0,1\}$, if s is consistent according to the following conditions.*

- *For an input node x_i, $s(x_i)$ is consistent with the input $x_0x_1\ldots x_{n-1}$. For an input wire $w = (x_i, y)$, $s(w)$ must equal $s(x_i)$.*
- *Non input nodes and edges have the values consistent with the input values and the labels of the nodes. For example, for a node, v, labelled by $\wedge$ with state of input wires $s(w_1) = 1, s(w_2) = 1$, $s(v)$ must equal 1.*

Definition 2 *A wire w (node v) is said to have switched from state s_0 to state s_1 if $s_0(w) \neq s_1(w)$ ($s_0(v) \neq s_1(v)$).*

We define a measure of energy consumption for a circuit. When a circuit C is subjected to an input $\vec{x}$, let wire w_i switch k_i times before the circuit is settled. Let l_i be the length of a wire w_i in the circuit C's embedding in a grid. Then the energy consumption for circuit C, $E_W(C, s, \vec{x})$ in state s with $\vec{x}$ as the input is defined to be $u \times \sum_{w_i \in W} k_i \times l_i$. For the Uniswitch model $k_i \leq 1$. We will distinguish between worst case energy consumption and average case energy consumption.

Definition 3 *The worst case energy consumption for a circuit C, $E_w(C)$, is defined to be $max_{s,\vec{x}} E_W(C, s, \vec{x})$, where max is taken over all (state, input vector) pairs.*

We similarly define the average case energy consumption. Note that an input assignment $\vec{x}$ defines the state of the circuit completely.

Definition 4 *The avarage case energy consumption for a circuit C is defined to be its energy consumption averaged over all initial states and all input vectors. Thus $E_a(C) = \sum_{s,\vec{x}} E_W(C, s, \vec{x}) / 2^{2n}$, where n is the number of input bits.*

For further details of this model, the reader is referred to Kissin's papers [4], [5].

3 Normal Transitive Functions

In this section, we show that for a general implementation of a normal transitive function (defined below), the average case energy $E_a(C)$ is $\Omega(n^2)$. The class of normal transitive functions encompasses all the functions that were used as examples of transitive functions by Vuillemin [10]. We first show that for a normal transitive function, $E_a(C)$ is no less than the switching energy consumed by $n/2$ edge disjoint paths from input ports to the output ports. Then we show that the energy consumed by these $n/2$ paths is $\Omega(n^2)$.

We first define a class of functions analogous to Vuillemin's transitive functions.

Definition 5 *A boolean function $f(x_1, x_2, \ldots, x_n, c_1, c_2, \ldots, c_k) = (y_1, y_2, \ldots, y_n)$ is said to be normal transitive, if it computes either the permutation group generated by the cycle (1 2 3 ... n) or a product of such groups. The bits $c_1, c_2, \ldots, c_k$ are the control bits specifying a permutation group element.*

In other words, the *normal transitive functions* embed a shifting like computation as a specialized instance. It is easy to verify that all the functions Vuillemin shows to be transitive, are just normal transitive. All his proofs show a reduction to the shifting permutation group. In particular, shifts, cyclic shifts, multiplication, convolution, linear transform and three matrix multiplication are all normal transitive functions.

We exploit the structure of the shift permutation group to show that when each of the n input bits and c control bits switch independently with probability 1/2 each, the expected number of output bits that switch is $n/2$.

Theorem 1 *For a normal transitive function, on average $n/2$ output bits switch.*

PROOF SKETCH: We prove the result for a shifting permutation group. It can be easily extended to a product of two shifting permutation groups of different order.

Let x_j denote the jth input bit and let y_l denote the lth output bit. Without loss of generality, assume that at time t, the control bits specify the identity permutation, *i.e.*, $y_j(t) = x_j(t), 1 \leq j \leq n$. Since f computes the shift permutation group, for any $\pi \in G$ such that the encoding of π differs from encoding of the identity permutation by $k/2$ bits, $y_{(j+k) \bmod n}(t+1) = x_j(t+1), 1 \leq j \leq n$, for some $0 \leq k \leq n-1$. In other words, each input bit is connected to an output bit k bits up/down. $n/2$ of n input bits are expected to switch. There are two cases.

1. $x_j(t) = x_{(j+k) \bmod n}(t)$. In this case, $x_j(t) = y_{(j+k) \bmod n}(t)$. The probability that $x_j(t)$ does not equal $x_j(t+1)$ is $1/2$. This implies that the probability that the bit $y_{(j+k) \bmod n}(t) \neq y_{(j+k) \bmod n}(t+1)$ is $1/2$.

2. $x_j(t) \neq x_{(j+k) \bmod n}(t)$. This case is similar to the previous one. Here, $x_j(t) \neq y_{(j+k) \bmod n}(t)$. The probability that $x_j(t)$ does not equal $x_j(t+1)$ is $1/2$. Once again, this means that the probability that the bit $y_{(j+k) \bmod n}(t) \neq y_{(j+k) \bmod n}(t+1)$ is $1/2$.

This shows that the probability of each output bit switching independently is $1/2$. Hence the number of output bits expected to switch is $n/2$.

□

The previous theorem shows that half of n output bits are expected to switch, even when we allow the permutation group element to change. Let $\mathcal{I}$ and $\mathcal{O}$ be the sets of input and output bits respectively. A partition of the chip is denoted by $p = (\mathcal{I}_L, \mathcal{I}_R, \mathcal{O}_L, \mathcal{O}_R)$, where $\mathcal{I} = \mathcal{I}_L \cup \mathcal{I}_R$ and $\mathcal{O} = \mathcal{O}_L \cup \mathcal{O}_R$. Let (i, o), where $i \in \mathcal{I}$, $o \in \mathcal{O}$, be an input bit position, output bit position pair. A pair (i, o) is said to be a *straddled pair* if either $i \in \mathcal{I}_L$ & $o \in \mathcal{O}_R$ or $i \in \mathcal{I}_R$ & $o \in \mathcal{O}_L$. There are $n/2$ (i, o) pairs that switch. In order to be able to show that most of the area of a chip switches, we have to show that a significant fraction of $n/2$ switching (i, o) pairs are straddled pairs.

Lemma 1 *For a normal transitive function, at least $n/3$ of n (i, o) pairs are straddled pairs.*

PROOF SKETCH: A partition could either divide $\mathcal{I}$ relatively evenly between two sides, *i.e.*, both $1/3 < |\mathcal{I}_L|, |\mathcal{I}_R| \leq 2/3$; or one of $\mathcal{I}_L$ or $\mathcal{I}_R$ has more than $2n/3$ bits. In the second case, it is easy to see that at least $n/3$ of the input bits from the side with $> 2n/3$ input bits will be paired with the output bits on the opposite side. In the first case, we average over all the n control word values. Let c denote the value of control bits $c_{\log n} \ldots c_2 c_1$. Let $\delta_{i,c}$ be 1 if x_i and y_{j+c} are on the opposite sides, and 0 otherwise. For every output bit y_j, there are at least $n/3$ input bits which are on the other side. Thus $\sum_{c=0}^{n-1} \delta_{j-c,c} \geq n/3$. If we sum it over all the output bit positions j, and divide it by the number of permutation group elements n, we find that an average group element creates at least $n/3$ straddled (i, o) pairs.

□

By Theorem 1, at least half of these straddled (i, o) pairs are expected to switch. Thus there are at least $n/6$ input bits that switch and whose corresponding output bits are on the opposite side. Next step is to show that these $n/6$ straddled pairs switch most of the area.

Theorem 2 *Let $f(x_1, x_2, \ldots, x_n, c_1, c_2, \ldots, c_{\log n}) = (y_1, y_2, \ldots, y_n)$ be a normal transitive function. The switching energy consumed by the switching of $\Omega(n)$ (i, o) pairs (x_{i_1}, y_{j_1}) $(x_{i_2}, y_{j_2}) \ldots (x_{i_{kn}}, y_{j_{kn}})$ is given by $\Omega(n^2)$.*

PROOF SKETCH: The proof is similar to the one given by Thompson to show that the area of a graph with bisection width ω is $\Omega(\omega^2)$. The bisection width of the subgraph spanned by $\mathcal{I}$ and $\mathcal{O}$ is k. We count the contribution to the lengths of the edges of k disjoint paths crossing a bisecting line $x = \alpha$ with one jog. Thompson counts the area contribution of the edges crossing such a line. □

We can generalize this result to the circuits where a wire is allowed to switch more than once. Then a circuit to compute a normal transitive function could have a bisection width $1 \leq k \leq n$. Each of the k edges in the bisection will have to multiplex $\Omega(n/k)$. We have to preclude the possibility that the chip sorts n/k bits in each group and sends them in that order, thus making the long wires switch only once. The following lemma shows that, on average, a n/k bit sequence has about $n/2k$ bit alternations. We assume that the sequencing of the bits in a n/k bits group is oblivious to the value of the input bits. We say that in a bit sequence $a_1, a_2, \ldots, a_l$, there is an alternation at the position j if $a_j \neq a_{j+1}$ for $1 \leq j \leq l-1$, *i.e.*, δ_j equals 1 if $a_j \neq a_{j+1}$ and 0 otherwise. The *total alternation* for a l-bit sequence is given by $\sum_{j=1}^{l} \delta_j$.

Lemma 2 *The average total alternation for a k-bit sequence from the set* $\{0,1\}^k$ *is* $(k-1)/2$.

PROOF SKETCH: Let the total alternation summed up over all the bit sequences in $\{0,1\}^k$ be denoted by $A(k)$. $A(k)$ is given by the following recurrence.

$$A(k) = 2A(k-1) + 2^{k-1}; \; A(1) = 0;$$

This equation has a solution in $A(k) = (k-1)2^{k-1}$. Averaged over 2^k k-bit sequences in $\{0,1\}^k$, we get an average total alternation of $(k-1)/2$.

□

By the preceding lemma, each of k edge disjoint paths will switch $\Omega(n/k)$ times. The following theorem shows that most of the area will switch every time.

Theorem 3 *The average switching energy of a normal transitive function computed by a circuit of bisection width k is bounded from below by* $\Omega(kn + n^2/k)$.

PROOF SKETCH: The proof to show that $\Omega(k^2)$ switching takes place at each time step is exactly similar to the proof of Theorem 2. When k is $O(1)$, just the storage requirements give $\Omega(n)$ switching energy. This adds up to $\Omega(k^2 + n)$ switching energy at each step. Lemma 2 assures that there will be $\Omega(n/k)$ time steps due to band width limitations. The lower bound of $\Omega(kn + n^2/k)$ is given by multiplying the lower bounds on area and time steps derived above.

□

The lower bound on the average switching energy of a normal transitive function derived in Theorem 3 attains a minimum of $n^{3/2}$ when the band width is $\sqrt{n}$.

4 Energy Complexity of Transitive Systolic Systems

In this section, by *Transitive Systolic System* we will refer to a system

- that is systolic.
- which computes a transitive function.

There are many versions of what it means for an algorithm to be systolic. We attempt to capture most of the commonly accepted characteristics of a systolic system in the following discussion.

1. *regular structures*: The most distinguishing characteristic of systolic systems is their regular structure. This makes them very suitable for VLSI implementations.

2. *few neighbors*: Each processing unit has a constant number of neighbors, which is independent of the input size n.

3. *pipeline*: These structures are capable of supporting a pipeline of input data at a regular pipeline period. Without loss of generality, we assume that at each time unit, a systolic system produces an output bit for each output data stream. The bubbles in a stream can be accommodated by defining a larger clock tick, within which the system is internally introducing these bubbles. Alternatively, one could distinguish between *bit systolic* and *word systolic* systems. A systolic system is *word systolic* if it produces a *word* of output at each clock tick. Similarly, a systolic system is *bit systolic* if it produces a *bit* of an output word at each clock tick. We insist on *word systolicity.*

4. *distribution of streams*: The input and output data streams should be evenly distributed around the periphery. This requirement is imposed by our desire to be able to use this structure in a larger system with compact routing. In an extreme case, if all the input and output data streams were tapped from the same convex side of the system, chances are that there will be a pitch mismatch between I/O side of this unit and I/O sides of other units overlayed together.

5. *nonzero delay*: Each stream has delay at least one in each processor.

We briefly describe what a transitive function is. We borrow the notation from Vuillemin [10].

Definition 6 *A boolean function* $f(x_1, x_2, \ldots, x_n, c_1, c_2, \ldots, c_k) = (y_1, y_2, \ldots, y_n)$ *is said to be transitive if:*

1. *For each value of the control input bits* $c_1, c_2, \ldots, c_k$ *the output vector* $(y_1, y_2, \ldots, y_n)$ *is the permuted input vector* $(x_{g(1)}, x_{g(2)}, \ldots, x_{g(n)})$*, where the permutation* g *is determined by the control bits.*
2. *The set of elements* g *forms a transitive permutation group.*
3. *For each pair* (i, j) *of index positions,* $1 \leq i, j \leq n$*, there exists a permutation* π*, such that* π *maps* i *into* j*, i.e.,* $\pi(i) = j$*.*

The main property of transitive functions is that each output bit depends on each input bit. Note that this makes transitive functions more restrictive than the functions for which each output value depends on each input value.

A systolic system may not exist for every transitive function. A function could be arbitrarily complex and still be transitive. An example is the union of a nonrecursive function and shift function, which is not even solvable. But, it still embeds an instance of a transitive computation. This result applies to only those transitive functions that are simple enough to have a systolic algorithm.

Before we go any further, we develop some notation. A data stream is said to be an input(output) data stream if the flow of data through this stream is into(out of) the system. Each input stream is incident on two convex faces of the convex embedding of the system. For a stream D, let $IF(D)$ denote the input face and let $OF(D)$ denote the output face.

Definition 7 *A description of a data stream D, $des(D)$, is a list $w_0\, pe_1\, w_1 \ldots pe_i\, w_i \ldots pe_n\, w_n$ of alternating wire segments and processors, such that wire segment w_{i-1} is incident into and wire segment w_i is incident out of the processor pe_i.*

Definition 8 *An input data stream D_i is said to dominate an output data stream D_o, denoted by $D_i \vdash D_o$, if both D_i and D_o are incident on a processor where bits in D_i are input to the computation of bits of D_o.*

Lemma 3 *For a transitive systolic function, every input data stream dominates every output data stream.*

REMARK 1: The description of a data stream D, $des(D)$ has a processor, wire segment pair (p_j, w_j), for every output data stream OD_i that D *dominates.*

Definition 9 *The span of an input data stream is defined to be the number of output data streams it dominates.*

Lemma 4 *In a transitive systolic system, when an input bit is switched, $\Omega(n)$ switching energy is consumed.*

PROOF SKETCH: Let us consider any input data stream, ID_i. Let E_i be the switching energy consumed when the data stream ID_i switches. We know from Lemma 3 that ID_i dominates each output data stream OD_j for $1 \leq j \leq m$. As we mentioned in Remark 1, this implies that $des(ID_i)$ has a processor, wire segment pair (p_j, w_j) for each of the output data streams OD_j. Thus, $des(ID_i)$ has at least $m = \Omega(n)$ wire segments.

According to our model, each wire segment has length $\Omega(1)$. Each wire segment w_k, when switched, consumes $e_k = \Omega(1)$ energy. When the input stream ID_i switches, each of the wire segments w_k in its description $des(ID_i)$ will switch. Thus, E_i, the energy consumed when ID_i switches is $\Omega(\sum_{w_k \in des(ID_i)} e_k)$. Since each e_k is at least constant and $des(ID_i)$ has $\Omega(n)$ wire segments, E_i is $\Omega(n)$.

□

Now we will prove a series of lemmas that lead us to our main theorem. In what follows, n is the number of input bits and m is the number of the output bits for the transitive function under discussion.

Lemma 5 *For a transitive systolic system, no input data stream can multiplex $o(k)$ input bit positions, for a constant c.*

PROOF SKETCH: The proof is based on the fact that for computing any output bit in a transitive function, every input bit is required. Each output bit requires n input bits. Thus each input data stream should provide one bit at each clock tick.

□

The key idea on which the proof of main theorem hinges, is that the area of a transitive systolic system is $\Theta(m \times n)$. Since m is $\Theta(n)$, this means that A is $\Theta(n^2)$. We first show that A is $\Omega(m \times n)$.

Lemma 6 *The area A of a transitive systolic system is bounded from below by $\Omega(m \times n)$.*

PROOF SKETCH: Notice that in any systolic system, no output bit stream can be multiplexed. Otherwise, there will be a constant buildup of information within the system. None of it can be discarded, since a transitive function needs it all. By Lemma 4, for a transitive function every input stream has a span of $\Omega(m)$. Thus each input stream induces wire length of $\Omega(m)$ in the system. By Lemma 5, there are at least $\Omega(n)$ such wires. This proves the result.

□

Now we are ready to show that A is $O(m \times n)$.

Lemma 7 *For transitive systolic functions, the area A of the system embedding is $O(m \times n)$.*

PROOF SKETCH: If a systolic system exists for a function, then an equivalent one can be realized as in the schema shown in Figure 1. The size of each processor P is a constant p. If it were not, then the system would not be systolic. Since the number of input bits and output bits is a constant, there is only a finite number of Boolean functions realizable. Each of these functions can be realized in constant area. This proves the result.

□

Theorem 4 *For transitive systolic systems, the average energy consumption is of the same order as the worst case energy consumption.*

PROOF SKETCH: Let $C = (V, W)$ be the circuit to realize a transitive function f with a systolic algorithm. The worst case (state, input) combination can make all of the area switch. Thus $E_w(C)$ is $O(A)$.

For a uniform distribution, each input bit x_i to the circuit can switch independently. The probability that a bit x_i switches P[x_i switches] = P[x_i goes $0 \rightarrow 1$] + P[x_i goes $1 \rightarrow 0$] = $1/2$. Thus the expected number of input bits switching is $n/2$. By Lemma 4 each switched input bit consumes $\Omega(n)$ energy. Thus a lower bound on $E_a(C)$ is $\Omega(n^2)$. From Lemmas 6 and 7, area A is $\Theta(n^2)$. This shows that $E_a(C)$ is $\Omega(A)$ and $E_w(C)$ is $O(A)$.

□

In the next section, we demonstrate this result with an example of a multiplier circuit. This algorithm is due to Wayne Winder of the VLSI Consortium at the University of Washington. This algorithm is of independent interest too, since it achieves the lower bound of n^2 on AP^2 for multiplication, as shown by Vuillemin [10].

4.1 A Multiplier Example

This multiplier is based on the basic shift and add paradigm of multiplication. The multiplication algorithm that we all learnt in grade school is shown in Figure 2. If instead of writing row i shifted left by $i - 1$ for $1 \leq i \leq m$, we wrote them all aligned, we will get this multiplication scheme. Each cell (i, j) receives a carryin from its north neighbor and a sum bit s_j from its northwest neighbor. It sends out the sum of $b_i \times a_j$, s_j and the carryin to its southeast neighbor, and carryout to its south neighbor. The last row is an exception. To get output bits $c_{n/2}$ on the right, we have an adder that sums up previous carrys with the sum bits. We can use an adder proposed by Brent and Kung [2]. This adder propagates the carry in time $O(\log n)$ and with width $O(n)$ and height $O(\log n)$. When pipelined, the adder has a period of $O(1)$. Thus the whole multiplier has area $O(n^2)$ and it works at a pipeline period of $\Theta(1)$. Thus its AP^2 is $O(n^2)$. Note that each input data stream has a total wire length equal to $\Omega(n)$. On average half the input bits switch, giving us a an avarage case energy consumption of $\Omega(n^2)$. The area of this circuit is $\Theta(n^2)$. The worst case switching energy is bounded by $O(A)$. This shows that this multiplier indeed consumes the same order of average case and worst case switching energy.

4.2 CFL Example

Just to demonstrate that it is not always true, we separate the average and worst case energy consumptions of a systolic implementation for recognizing a context free language $L = \{ww^R\} \cup \{1\{0,1\}^*\}$. We refer to the systolic implementation of Guibas, Kung and Thompson [3]. In the worst case, a string of the form ww^R where $w \in \{0,1\}^n$ dominates with an energy complexity of n^2. In the average case, though, the strings of the form $1\{0,1\}^{n-1}$ dominate, since there are $2^{2n-1} - 2^n$ of them compared to 2^n of ww^R kind.

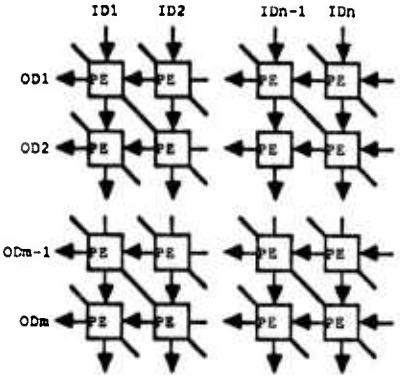

Figure 1: The Systolic System Schema

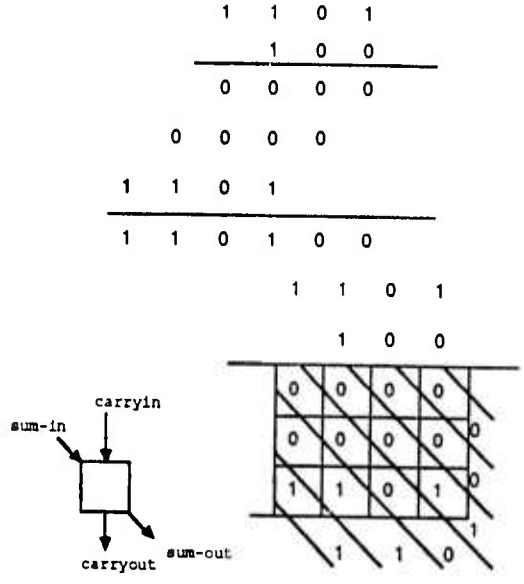

Figure 2: The Multiplication Algorithm

5 Lower Bounds

In this section, we extend the lower bound result of Kissin [4]. She shows that to compute a 1-switchable function in depth $O(\log n)$, worst case switching energy of $\Omega(n \log n)$ must be used. The assumption is that the output nodes are placed on a convex border in the embedding. We show that if the depth can be relaxed to be a polynomial in n, *i.e.*, $d(n) = n^\epsilon$ for $0 < \epsilon \leq 1$, then area $\times$ depth $= \Omega(\max(n \log n,\ n^{1+\epsilon}))$. Similarly, if the depth is polylog, *i.e.*, $d(n) = log^k n$ for $k \geq 2$, then area $\times$ depth $= \Omega(n\, log^k n)$. These lower bounds hold under the assumptions that in an embedding of the circuit C the output nodes are located on a convex border and the fan out is limited to two.

Next, we will define *1-switchable functions.* A function f is 1-switchable if there is an input bit position on which every output bit depends. A formal definition follows.

Definition 10 *A function $f : \{0,1\}^n \longrightarrow \{0,1\}^m$, with $m = \Theta(n)$, is said to be 1-switchable if there exist two input vectors $\vec{a} = x_1 x_2 \ldots x_k \ldots x_n$ and $\vec{b} = x_1 x_2 \ldots \overline{x}_k \ldots x_n$ differing only in one bit position, such that $\Theta(n)$ output bits switch from $f(\vec{a})$ to $f(\vec{b})$.*

Note that the class of transitive functions forms a subset of the class of *1-switchable* functions. Thus all the lower bounds that we derive also apply to transitive functions.

Our basic problem is to fan out the input bit x_k to $\Theta(n)$ output bit positions. If we restrict the depth for this fan out, then we can place lower bounds on the sum of the lengths of the edges in a Cartesian grid embedding. We limit ourselves to the embeddings (underlying communication graphs) with fan out of at most 2 for every vertex. Our strategy will be to first show that any efficient fan out graph will be in a canonical form, which we call *balanced fan out* graph. Then we show a lower bound on the area of any embedding for this graph, given a

certain depth. Note that since we are working with the unit delay model, depth corresponds to delay in the given function. A wire can be driven in unit time regardless of its length.

Definition 11 *Let $G = (V, E)$ be a directed, rooted graph. Let $v_0 \in V$ be the designated input node and $V_O \subseteq V$ be a designated set of output nodes Then $F = (V, E, v_0, V_O)$ is said to be a fan out graph if there exists a directed path in G from v_0 to each $v_k \in V_O$.*

We limit ourselves to only those embeddings in which the output nodes are located at a convex border. Without loss of generality, we can assume that all the output nodes are located along a straight line. This blows up the area at most by a constant factor. Thus we restrict the definition of a fan out graph to get an appropriate class of graphs which admit such convex embeddings. In a digraph $G = (V, E)$ let u be the predecessor of v, denoted by $pred(v)$, if the edge $(u, v) \in E$. In this case, v is called u's successor, denoted by $succ(u)$.

Definition 12 *A fan out graph $F = (V, E, v_0, V_O)$ is said to be a convex fan out graph if for every $v \in V_O$, its successor also is an output node, $succ(v) \in V_O$. In this case, all the nodes in $V - (\{v_0\} \cup V_O)$ are called internal nodes.*

Definition 13 *A convex fan out graph $F = (V, E, v_0, V_O)$ is said to be optimal if there is no other convex fan out graph from v_0 to V_O with depth less than equal to depth(F), and with an embedding with less area than minimum area embedding of F.*

REMARK 2: Any optimal convex fan out graph will be acyclic. Back edges do not fan out into a new node. There is a convex fan out graph with all the back edges removed, which has the same order of depth but less switching energy. Now on, we will assume that our canonical form graphs, balanced fan out graphs, are acyclic.

Definition 14 *A linear chain consists of an alternating node,edge list, $v_i\, e_i\, v_{i+1} \ldots v_{j-1}\, e_{j-1}\, v_j$, such that $e_k = (v_k, v_{k+1})$, for $i \leq k \leq j-1$, and every v_l, in the list has at most one outgoing edge. A linear chain is maximal if $pred(v_i)$ has two outgoing edges and either v_j is a leaf or $succ(v_j)$ has two outgoing edges. Node v_i is called the leader of the chain. The length of the chain is the number of nodes in its list.*

Lemma 8 *An optimal convex fan out graph does not have any maximal linear chains consisting of internal nodes.*

PROOF SKETCH: By contradiction. Assume that $F = (V, E, v_0, V_O)$ is optimal and it has a linear chain. $v_i\, e_i\, v_{i+1} \ldots v_{i+l} \ldots e_{i+k-1}\, v_{i+k}$ (note that if v_{i+l+1} is an output node, then no node $v_q, i \leq q \leq i+l$ can be an output node and no node $v_r, i+l+1 \leq r \leq i+k$ can be an internal node, since F is a convex fan out graph). We can get a new convex fan out graph, F', by deleting nodes v_i through v_{i+l} and edges e_{i-1} through e_{i+l} and by inserting the edge $(pred(v_i), v_{i+l+1})$. Note that F' has depth at most equal to the depth of F and its switching energy has been reduced by at least $l+1$. □

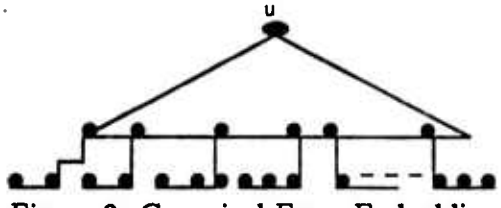

Figure 3: Canonical Form Embedding

NOTE: A linear chain consisting of only output nodes is called a *terminal chain.*
Thus , an optimal convex fan out graph will consist of a binary tree stage whose leaves each feed a linear chain, as shown in Figure 3. It is easy to see that, for optimality, the binary tree must be a balanced binary tree. It will be a complete binary tree if the number of leaves is an integral power of two.

Definition 15 *A set of linear chains is balanced if every chain has length k or $k+1$ for some integer k.*

Lemma 9 *For an optimal convex fan out graph, the set of its terminal chains is balanced.*

This defines our family of acceptable fan out graphs. A fan out graph F is said to be a *balanced fan out graph* if it is a convex acyclic fan out graph with balanced terminal chains.

Theorem 5 *Let $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ be a 1-switchable function. Let $C(V,W)$ be a circuit to compute f. For any convex embedding of C within depth $d(n)$, $log^2 n \leq d(n) \leq n^\epsilon$, with $0 < \epsilon \leq 1$, the following holds. The $E_w(C) \times d(n)$ is $\Omega(\max(n \log n,\ n\, d(n))$.*

PROOF SKETCH: The proof relies on the result of Brent and Kung on the minimum area required for a convex embedding of a binary tree [1]. The terminal linear chains along a line contribute $\Omega(n)$ to the sum of edge lengths, and consequently to area and switching energy. The second factor of $n/d(n) \times log\,(n/d(n))$ is from the balanced binary tree embedding.

□

6 Acknowledgements

We are indebted to the staff of the University of Washington VLSI Consortium in many ways, particularly Larry McMurchie and Wayne Winder. We thank Wayne Winder for letting us use his multiplication algorithm as an example.

References

[1] R.P. Brent and H.T. Kung. The Chip Complexity of Binary Arithmetic. In *ACM Symposium on Theory of Computing*, ACM-SIGACT, 1980.

[2] R.P. Brent and H.T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, March 1982.

[3] L. J. Guibas, Kung H. T., and Thompson C. D. Direct VLSI Implementation of Combinatorial Algorithms. In *Cal Tech Conference on VLSI*, Cal Tech, 1979.

[4] G. Kissin. Functional Bounds on Switching Theory. In *Chapel Hill Conference on VLSI*, U.N.C., Chapel Hill, Computer Science Press, 1985.

[5] G. Kissin. Measuring Energy Consumption in VLSI Circuits: a Foundation. In *ACM Symposium on Theory of Computing*, ACM-SIGACT, 1982.

[6] T. Lengauer and Mehlhorn K. On the Complexity of VLSI Computations. In *Proceedings of CMU Conference on VLSI*, CMU, Computer Science Press, 1981.

[7] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[8] L. Snyder and A. Tyagi. *The Energy Complexity of Transitive Functions.* Technical Report TRCS-86-09-07, Dept. of Computer Science, University of Washington, Seattle, 1986.

[9] C.D. Thompson. Area-Time Complexity for VLSI. In *ACM Symposium on Theory of Computing*, ACM-SIGACT, 1979.

[10] J. Vuillemin. A Combinatorial Limit to the Computing Power of VLSI Circuits. In *IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, 1980.

Energy Complexity and Delay Comparison of Dynamic and Static PLA Design Styles†

Akhilesh Tyagi*
Department of Computer Science
University of Washington
Seattle, WA 98195

†Submitted to the Stanford Conference on Advanced Research in VLSI, 1986.

* Supported in part by DARPA under Contract MDA 903-85-K-0072.

Energy Complexity and Delay Comparison of Dynamic and Static PLA Design Styles

AKHILESH TYAGI
Department of Computer Science
University of Washington
Seattle, Washington

ABSTRACT

In this paper, we compare the energy complexities of the static and dynamic design styles for a PLA. We show that the average energy consumption of a dynamic PLA exceeds that of a static PLA. We also show that, on average, a dynamic PLA is faster than a static PLA. This is consistent with intuition since one can trade power for delay. In order to prove these results, we deal with a very general class of PLAs. If we are allowed to restrict this class, we can do more. In particular, we show that the choice of a design style for a data path control PLA depends on the degree of parallelism of the data path. Our techniques are general enough to be applicable to most of the application domains. We believe that our results give a mathematical justification (within the limitations of our model) for picking one design style over the other for a given application domain.

1 Overview

Due to its regularity, a PLA is probably the most popular structure for the implementation of random logic. Its simplicity also makes it the ideal medium for automating the layout process for a random logic expression. Recently, the PLA has been the structure of choice for the control path design in microprocessors, as in QuarterHorse [5], Mosaic [9], and PP4 [10], primarily due to the ease of reconfiguring it. Thus, a significant fraction of silicon area laid out today, is devoted to PLA design. It is no surprise, then, that a lot of effort has been devoted to finding ways of optimizing a PLA

at the description level (*e.g.* logic minimization [2]) as well as at layout level (*e.g.* folding techniques [4], [3]). However, to the best of the author's knowledge, there have been no attempts at mathematically analyzing the energy consumption or the delay of this structure.

With ever increasing demand for performance on the single chip VLSI architectures, even a CMOS design needs to be energy conscious. The power in the CMOS technology is a direct function of its clock frequency. The high performance architectures with the integration approaching submicron levels are reaching a point where the power dissipation will turn out to be a problem to contend with. Paying attention to the switching complexity of a PLA is specially fruitful, since it is the control path in an architecture which does most of the switching.

In Section 2 we define a model based on energy consumption. We also refine the original VLSI model [11] to take into account the mobility differences between p-type and n-type channels and the strength of a device driving a node, for the delay calculations. Then we compare two design styles under the energy complexity measure in this model for a general class of PLAs and the datapath control PLAs. In the last section we compare the delay in the two design styles.

2 Model

Our energy model is the same as the one originally outlined by Kissin [8]. The VLSI model is the commonly accepted one, proposed by Thompson [11]. A layout can be viewed as an embedding of the communication graph in a Cartesian grid. Each grid point can either have a processor or a wire passing through. A wire can not go through a grid point unless it is a terminal of the processor at that grid point. Wires have unit width and bandwidth and processors have unit area. The initial data values are localized to some constant area, to preclude an encoding of the results.

2.1 Energy Model

Let u be the unit switching energy defined to be the energy spent when a wire of unit length switches (changes state either from a $1 \rightarrow 0$ or from a

$0 \rightarrow 1$). Note that a minimum size transistor also consumes $O(u)$ energy when it changes state. We will say that a wire of length l takes $O(l)$ energy when it switches. It is implicit here that u is a technology dependent constant.

We do not account for the switching energy consumed by the transistors. This is in accordance with the widely held belief that the wires take up most of the area in a VLSI layout. In asymptotics, the wires dominate the complexity of even a regular structure like a PLA. In a static design, it may seem that when a pull-up is fighting against one or more pull-downs, a considerable amount of energy may be consumed, thus violating this assumption. But, in a static design the pull-up transistors are so weak that the energy consumed by the contention of a pull-up and a pull-down device is a very small fraction of the switching energy of a long minterm[1] wire in poly layer. The switching energy of a wire is proportional to its length in the layout.

A PLA satisfies the Uniswitch Model (USM), as defined in Kissin [8]. In this model, any wire in an acyclic circuit can switch at most once, for unpipelined computation. A PLA does not switch any wire more than once in response to a clock transition, since the state feedback paths (to build a finite state machine) are clocked. A symmetric notion is that of Multiswitch Model. Kissin shows some upper bounds on the switching energy of certain circuits in the Multiswitch model.

Assumptions:

1. Each node in the PLA depends on an input, *i.e.*, for each node, a pair of inputs exists which makes that node switch.

2. Inputs are synchronous. In other words, they are presented at the input ports, simultaneously. The inputs in a PLA have to be clocked, for it to work as a component of a larger system.

We can abstract a PLA circuit as a graph, $C(V, W)$, where V is the set of nodes and W is the set of edges or wires in $V \times V$.

[1] Although the correct term is *implicant*, we will continue to use *minterm* due to its wide acceptability

Definition 1 *A PLA circuit, $C(V,W)$, is said to be in state $s : V \cup W \longrightarrow \{0,1\}$, if s is consistent according to following conditions.*

- *For an input node x_i, $s(x_i)$ is consistent with the input $x_0 x_1 \ldots x_{n-1}$. For an input wire $w = (x_i, y)$, $s(w)$ must equal $s(x_i)$.*
- *Non input nodes and edges have the values consistent with the input values and the labels of the nodes. For example, for a node, v, labelled by $\wedge$ with state of input wires $s(w_1) = 1, s(w_2) = 1$, $s(v)$ must equal 1.*

Note that the state of a PLA is completely specified by the value of its inputs. Over a complete cycle of operation, it defines the final values of the minterms and the outputs.

Definition 2 *A wire w (node v) is said to have switched from state s_0 to state s_1 if $s_0(w) \neq s_1(w)$ ($s_0(v) \neq s_1(v)$).*

We define a measure of energy consumption for a circuit. When a circuit C is subjected to an input $\vec{x}$, let wire w_i switch k_i times before the circuit is settled. Let l_i be the length of a wire w_i in the circuit C's embedding in a grid. Then the energy consumption for circuit C, $E_W(C, s, \vec{x})$ in state s with $\vec{x}$ as the input is defined to be $u \times \sum_{w_i \in W} k_i \times l_i$. For the Uniswitch model $k_i \leq 1$. We will distinguish between worst case energy consumption and average case energy consumption.

Definition 3 *The worst case energy consumption for a circuit C, $E_w(C)$, is defined to be $\max_{s,\vec{x}} E_W(C, s, \vec{x})$, where the maximum is taken over all (state, input vector) pairs.*

We similarly define the average case energy consumption. Note again that for a PLA, an input assignment $\vec{z}$ defines the state of the circuit completely.

Definition 4 *The average case energy consumption for a circuit C is defined to be its energy consumption averaged over all initial states and all input vectors. Thus $E_a(C) = \sum_{s,\vec{x}} E_W(C, s, \vec{x}) \,/\, 2^{2n}$, where n is the number of input bits.*

For further details of this model, the reader is referred to Kissin's papers [7], [8].

We compare only the total amount of switching between two design styles, thus separating the inherent complexity issue from the layout issues. The clever layout techniques like folding, to reduce the wire lengths could very well be applied across both the design styles.

We assume that the domain of PLAs we are considering has the property that, given s_i is the current state, the next state could be any of the 2^n states with equal probability. Of course, in a real PLA certain state transitions are more heavily favored than the others. We do not have sufficient information about a PLA personality to assign nonuniform weights to the state transitions. A more complicated modelling could attempt to look at it as a Markov process. Alternatively, one could model a PLA for a certain domain of applications. In Section 4, we model a control PLA for a data path. We have also assumed that no two states share a set of minterms, in other words, there are no *don't care* bits in the *and plane*. If there are *don't care* bits in the *and plane*, it just reduces the state space size, and hence the amount of total switching.

2.2 Delay Model

For our puposes, we can work with any of the *synchronous, capacitive, or diffusion* model for the delay calculation as described in Bilardi, Pracchi, Preparata [1]. However, we have to refine our model to include, in our delay calculations, the *strength* and the *type* of a driver driving a node. The following description justifies the need for this refinement.

Dynamic Style: An essential difference between static and dynamic PLA design styles is that the dynamic style assigns the pull-up and pull-down phases for the same node to separate phases of a clock signal. Thus, a node is never pulled down and up, simultaneously. Let the time a minimum sized n-channel driver takes to pull down a unit area metal wire, be denoted by ν. We will assume that the time it takes a minimum sized n-channel driver to discharge a minterm wire in a given PLA instance is given by ν, since the scale factor consisting of

minterm layer capacitance and area is same for both the styles. Thus for the sake of comparison, our results will still be valid. We will not charge any time for a $0 \rightarrow 1$ transition in a dynamic design, since the *precharge* phase can be made very very small compared to the *evaluate* phase. If it is a $1 \rightarrow 0$ transition, then the strength of this transition is defined to be the number, k, of n-channel devices pulling it down. This transition is charged a delay of ν/k. In the SPICE experiments we performed on some real PLAs, this relationship seems to hold.

Static Style: In a static design, when a node is to be pulled down to ground, there is a weak p-channel device fighting against one or more n-channel devices. The p-channel device is designed to be weak enough so that it can be overpowered even by a minimum sized n-channel device. Typically, the channel length ratio for a p-channel device to a n-channel device is about two, in a static PLA design (Note that this ratio is required in order to have reasonable noise immunity, as shown in the text book by Weste and Eshraghian [12] [page 54]). In the SPICE experiments conducted by the author, the time a n-channel device took to discharge a node without a weak p-channel device pulling it up, was almost the same as the time in a static setup. Thus, we assume that a minimum sized n-type device takes time ν to pull down a minterm node even in the static design style. The time it takes for a p-channel device to pull up the same node is different due to lower mobility of a p-channel. Let μ_r be the ratio of n-channel mobility to p-channel mobility, *i.e.*, $\mu_r = \mu_n/\mu_p$. Let us denote the time, $\mu_r\nu$, a minimum sized p-channel transistor takes to pull up a minterm by ν_p. Then a p-channel device with a channel width ratio of r, will take time $r\nu_p$. This case, when a weak p-type pulls up a node, shows the main difference between two design styles. As we previously mentioned, this time equals $r\mu_r\nu$, where r here is about two, and μ_r in a typical process today varies between two and three. Thus a $0 \rightarrow 1$ transitions takes ~ 5 times as long as a slowest $1 \rightarrow 0$ (strength 1, when only one n-channel device is on).

To summarize the delay estimation, there is a delay of ν/k, when a node is pulled down with strength k. There is a delay of $\sim 5\nu$, when a node is

pulled up in a static setup. We allow a node to be pulled up with zero delay in a dynamic setup.

3 Energy Complexity

In what follows, m is the number of minterms, n is the number of inputs, and, l is the number of outputs. Thus the total area of a PLA under either implementation, is $\Theta(m\,(n+l))$. Note that the input switching is uniform to both the design styles, however the way the minterms and the outputs are evaluated is very different. We demonstrate our results only for minterm evaluation. Also notice that the energy consumption for both the NAND and NOR dynamic styles is equal. The amount of charge to be placed on a wire or to be carried away from a wire remains the same. The delay through a NAND style PLA, of course, is longer than the delay through a NOR style PLA. The same techniques apply to the output evaluation and consequently tilt the balance in the same direction. We develop some notation before we go any further. Let S be the set of minterms. More specifically, S has elements in the range $[0, m-1]$.

Definition 5 *An onset for a state s_i, denoted by $On(f, s_i) \subseteq S$ is the set of minterms that are active (on) in the state s_i for a PLA for the function f.*

Let $\sim On(f, s_i)$ denote the bitwise complement of $On(f, s_i)$. We will think of $On(f, s_i)$ as a bit vector of length m. Thus $\vec{a} \cap (\cup)\vec{b}$ denotes the bitwise '$\wedge(\vee)$' of $\vec{a}$ and $\vec{b}$. The cardinality of a bit vector $\vec{a}$, $|\vec{a}|$ is the number of bit positions with entry 1 in $\vec{a}$.

We now analyze the switching of minterms in the static and dynamic design styles.

3.1 Static Design Style

Let us consider the switching for a transition from a previous state, s_p, to a current state, s_c. There are two components to switching in a static design.

1. The previous state component, which requires that the minterms that were *on* in the previous state and are *off* in the current state be turned off. The number of such minterms is $|(On(f, s_p) \cap (\sim On(f, s_c)))|$.

2. The current state component, which requires that the minterms that were *off* in the previous state and are *on* in the current state be turned on. The number of such minterms is $|((\sim On(f, s_p)) \cap On(f, s_c))|$.

Thus, the total switching, $Sw(s_p, s_c)$, from the state s_p to s_c is given by:

$$Sw(s_p, s_c) = |(On(f, s_p) \cap (\sim On(f, s_c)))| + |((\sim On(f, s_p)) \cap On(f, s_c))| \quad (1)$$

In order to maximize $Sw(s_p, s_c)$, for each pair (s_p, s_c), $On(f, s_c)$ must equal $\sim On(f, s_p)$. In that case, all of the m minterms will switch in going from any state s_p to any other state s_c. It is easy to verify that to attain such a condition m should be exponentially large in n. In such a situation, a PLA is not a structure of choice. A ROM would be the structure to use. Thus, we rule out the complete switching for a reasonable PLA structure. In practice, PLAs are designed with m being equal to $k\,n$, where k is a very small constant.

We start with the assumption that m equals n. We will generalize it as we go on. The expression in Equation 1 is symmetric with respect to the pair (s_p, s_c). That suggests that we could count the overlaps between all the subsets of S and double it to get the switching totalled over all pairs (s_i, s_j). This is almost true. There is another additional term due to self overlap. Note that the total number of unordered pairs (s_i, s_j) is $p = 2^n(2^n - 1)/2$.

Lemma 1 *The average switching for a static PLA design with $m = n$ is given by $n\,2^{n-1}/(2^n - 1)$.*

PROOF: Each column in the truth table for n variables has 2^{n-1} 1's. Thus, each column contributes $(2^{n-1} - 1)(2^{n-1})/2$ to the total overlap. Thus total overlap is $n(2^{n-1} - 1)(2^{n-1})/2$. There is another term of $n\,2^{n-1}$ contributed by the overlap of complement pairs. Thus the total switching is $2\,n(2^{n-1} - 1)(2^{n-1})/2 + n\,2^{n-1}$. This equals $n\,2^{2n-2}$. The average switching then is $n\,2^{2n-2} / 2^{n-1}(2^n - 1)$. This number tends to $n/2$ in the limit.

□

We illustrate the previous lemma by an example. Consider the following table of minterm values for $m = 2$.

m_0	0	0
m_1	0	1
m_2	1	0
m_3	1	1

Consider the transitions from m_0 to any other state. m_3 is m_0's complement. The overlap between m_0 and m_i for $1 \leq i \leq 3$ gives the total switching with initial state m_0 and next state m_j for $0 \leq j \leq 2$. These cases are taken care of by the term $(2^{n-1} - 1)(2^{n-1})/2$. If the next state is the complement of the current state then the switching just equals the number of ones in the current state vector. Summed over each possible current state this overlap equals the number of ones in the table, which is given by $n\,2^{n-1}$.

The next generalization comes from considering the case when $k > 1$. For each one of 2^n input states, we need to assign a distinct minterm from a set of $2^m > 2^n$ minterms, (We assume that there is no sharing of minterms). Let M be the set of the 2^n minterms assigned. There are two cases, one when M is closed under bitwise complement, and two, when it is not. The average switching for case one seems to be lower than for the case two, because of the symmetry of the expression in Equation 1. Let us consider the first case. There are 2^{n-1} of complemented pairs in M. Intuitively, it seems like that given a large number 2^m of minterms to choose from, we could intelligently pick 2^n minterms to lower the switching below Lemma 1 level. But as the following theorem shows, the switching only gets worst. The intuitive reasoning behind this phenomenon is as follows. Each complement pair has m 1's between two of them. There are 2^{n-1} such pairs. Thus the total number of 1's in the elements of M is $m\,2^{n-1}$. The total number of bits in these entries is $m\,2^n$. There are exactly half 1 entries and half 0 entries. Thus, the expected Hamming distance between any two elements of M is $m/2$. We state and prove the theorem next.

Theorem 1 *Let the number of minterms $m = kn$ with $k > 1$. Let the set of assigned minterms M be closed under bitwise complement. Then the average switching between any two states, $Sw(s_i, s_j)$, is given by $m\,2^{n-1}/(2^n - 1)$.*

PROOF: The proof is similar to the one given for Lemma 1. Notice that the overlap between two complement pairs is exactly m. There are 2^{n-1} complement pairs. Thus, the total number of distinct unordered pairs of complement pairs is given by $(2^{n-1}-1)(2^{n-1})/2$. In addition, each pair has a self overlap of m. Thus, the total switching is $2\,m(2^{n-1}-1)(2^{n-1})/2 + m\,2^{n-1}$. Dividing it by $2^{n-1}(2^n - 1)$ gives the average switching to be $m\,2^{n-1}/(2^n - 1)$. Note that the average switching tends to $m/2$ in the limit.

□

Now we consider the second case when the set M is not closed under complement. This gives rise to a range for the average switching depending on the number of unpaired minterms, $2r$, in M.

Theorem 2 *Let $m = kn$ with $k > 1$. Let there be $2r$ unpaired minterms in M. Then the average switching $Sw(s_i, s_j)$ is given by*

$$\frac{m\,(2^{2n-2} - r^2)}{2^{n-1}\,(2^n - 1)} + (m-1)\left(\frac{2^{m-2}}{2^{m-1}-1}\right)\frac{4r^2 - 2r}{2^{n-1}(2^n - 1)}.$$

PROOF: We have to consider the interactions between three groups of the minterms belonging to M. Let set A consist of the minterms that have their complements in M. The cardinality of A is $2^n - 2r$. Let $2r$ unpaired minterms belong to set B. Let the bitwise complements of $2r$ minterms in B belong to set C. There are three terms in the expression for the total switching.

- First one is due to internal overlap between the elements of A. This analysis is similar to the one in the proof of Theorem 1. This term contributes $m(\,(2^{n-1} - r)(2^{n-1} - r - 1)\,)$. There is a self overlap term of $m(2^{n-1} - r)$.

- This term is due to the interaction between set A and the sets B and C, due to the symmetry of Expression 1. Its contribution is $(2^{n-1} - r)2r$.

- This term approximates the contribution of the interaction of sets B and C. Note that for each pair of bit vectors $\vec{u}, \vec{v} \in B$, we need to calculate $|\vec{u} \cap \sim \vec{v}| + |\sim \vec{u} \cap \vec{v}|$. Also Notice that this quantity equals exactly the number of bit positions $\vec{u}$ and $\vec{v}$ differ in. This is the Hamming distance of $\vec{u}$ and $\vec{v}$, denoted by $h(\vec{u}, \vec{v})$. Thus, this term's contribution to the total switching is $H = \sum_{\vec{u},\vec{v} \in B} h(\vec{u}, \vec{v})$, which is the Hamming distance of the set B. If the pairs $(\vec{u}, \vec{v})$ and $(\vec{v}, \vec{u})$ are counted as distinct pairs, then H is known as the ordered Hamming distance of B, otherwise it is the unordered Hamming distance. We can not really estimate H, the Hamming distance of a set for an arbitrary set B. However, since we are interested in the average behavior, we can calculate the average Hamming distance of a set picked from a given collection of sets. In this case, all the sets in this collection must not be closed under complement. The largest such collection consists of lexicographically first 2^{m-1} m-bit vectors, from the complete enumeration of m-bit vectors, *i.e.*, the set $\{0\,\{0,1\}^{m-1}\}$.

  First of all, note that the Hamming distance of a set taken over all ordered pairs $(\vec{u}, \vec{v})$ (Ordered Hamming Distance) is exactly twice the Hamming distance taken over all unordered pairs (Unordered Hamming Distance), due to symmetry of the relation $\cap$. For a given set, calculating its ordered Hamming distance is much easier than calculating its unordered Hamming distance. For each bit position, a bit vector $\vec{u}$ differs from $2^m - 1$ vectors. Thus, the ordered Hamming distance of the set of 2^m distinct m-bit vectors is $m2^{m-1}2^m$ and its unordered Hamming distance is $m2^{m-1}2^m / 2$. We are interested in the unordered Hamming distance of lexicographically first 2^{m-1} vectors out of 2^m distinct m-bit vectors. Note that the first column entries for all of 2^{m-1} vectors is zero. Thus the first column does not contribute anything to the Hamming distance of this set. The remaining columns contribute exactly as much as the Hamming distance of the complete set of $(m-1)$-bit vectors. Thus the total Hamming distance we are looking for is given by $(m-1)2^{2m-4}$. It has to be averaged

over $2^{m-2}(2^{m-1}-1)$ pairs. This gives us an average overlap of

$$\frac{(m-1)2^{m-2}}{(2^{m-1}-1)}\,.$$

Thus, the total contribution of this term is

$$\frac{(m-1)2^{m-2}(2r)(2r-1)}{(2^{m-1}-1)}\,.$$

All the three terms combined and averaged over $2^{n-1}(2^n-1)$ pairs give an average switching equal to

$$\frac{m\,(2^{2n-2}-r^2)}{2^{n-1}\,(2^n-1)} \;+\; (m-1)\left(\frac{2^{m-2}}{2^{m-1}-1}\right)\frac{4r^2-2r}{2^{n-1}(2^n-1)}\,.$$

□

The first term

$$\frac{m\,(2^{2n-2}-r^2)}{2^{n-1}\,(2^n-1)}$$

dominates as $2r/2^n$ tends to zero, and the average switching tends to $m/2$. As the ratio of unpaired minterms rises, the second term dominates in the average switching expression. Thus, for $2r/2^n$ tending to 1, the second term

$$(m-1)\left(\frac{2^{m-2}}{2^{m-1}-1}\right)\frac{4r^2-2r}{2^{n-1}(2^n-1)}$$

tends to

$$(m-1)\frac{2^{m-1}}{2^{m-1}-1} \;\sim\; m\,.$$

Thus, the average switching is a monotonic nonlinear function of $2r/2^n$ with its minimum at $m/2$ and the maximum at $\sim m$.

3.2 Dynamic Design Style

The general expression for the switching, $Sw(s_p, s_c)$, is different in the precharged logic. In this expression, there is no coupling between two states s_p and s_c. Once again, the two components in this case are:

1. Precharging the minterms that were discharged in the previous state. This number is $m - |On(f, s_p)|$.

2. Discharging the minterms that are not active in the current state. This number is $m - |On(f, s_c)|$.

Thus, the expression for total switching in the dynamic logic case is given by:

$$Sw(s_p, s_c) = 2m - |On(f, s_p)| - |On(f, s_c)| \quad (2)$$

Notice that the worst case switching in this case could be almost as large as $2m$, while in the static case the worst case switching is limited to m.

Let us consider the case when m equals n.

Lemma 2 *The average switching for a dynamic PLA design with $m = n$ is n.*

PROOF: This time, we have to sum $|On(f, s_p)| + |On(f, s_c)|$ over all unordered pairs (s_p, s_c). Each row is counted $2^n - 1$ times. This sum taken over all the rows in the truth table equals $2^n - 1 \times$ *(total number of 1's in the truth table)*. Dividing it by the number of unordered state pairs gives $n\,2^{n-1}(2^n - 1)$ / $2^{n-1}(2^n - 1)$, which equals n. According to Equation 2, the average switching is this number subtracted from $2n$, which is n.

□

Contrast this with the average switching for the static case which is $n/2$ (in the limit). Again, the reason for that is the independence of two states in the dynamic case. In other words, a dynamic design has no memory.

The current state s_c does not remember, and hence does not benefit from, anything about the previous state, s_p.

Let us consider the case when $m = kn$ for $k > 1$. Once again, the first case we consider is when M is closed under complement.

Theorem 3 *Let the number of minterms $m = kn$ with $k > 1$. Let the set of assigned minterms M be closed under bitwise complement. Then the average switching between any two states, $Sw(s_i, s_j)$, is m.*

PROOF: Once again, each row is counted $2^n - 1$ times. Each pair of complemented minterms contributes m 1's to the truth table. Thus the total number of 1's in the truth table is $m\,2^{n-1}(2^n - 1)$. This term averaged over all unordered state pairs gives m. Thus, the average switching equals m.

□

In the second case, the average switching turns out to be m, again.

Theorem 4 *Let there be $2r$ unpaired minterms in M. Then the average switching $Sw(s_i, s_j)$ is given by m.*

PROOF: The number of complement pairs is given by $2^{n-1} - r$. To count the number of 1's in the unpaired $2r$ minterms, we make the assumption that the number of 1's in a minterm is equal to the expected number of 1's in a minterm. Since, exactly half the entries in a truth table are 1, this expected number is $m/2$. Hence, the sum $|On(f, s_p)| + |On(f, s_c)|$ equals $(2^n - 1)((2^{n-1} - r)k + (2rm/2))$. This leads to an average switching of m.

□

4 Data Path PLA

In the preceding discussion, we considered the PLAs that have the complete sets of m-bit vectors as the minterm set. We also assumed that from a given

state, it was equally likely to go to any other state in the next transition. Given all these assumptions, we derived some general average case results. Can we do better if we have more information about the domain of PLAs we analyze? We answer this question with regard to a very interesting and important class of PLAs, the data path control PLAs. We believe that most of the PLAs designed today are used as finite state machines (FSM).

To be able to get a handle on the structure of the minterm set M, we will make some simplifying assumptions about the behavior of the data path control FSMs. Note that it does not have to be a data path control PLA. Even for most other control applications, the following assumptions hold.

1. A data path consists of many components like a shifter, ALU, register file and PC. Each such component has a cluster of control lines. When a component is logically activated, all the control lines associated with it become active over a macrocycle of the FSM. There are smaller microcycles, probably a clock cycle, which define the granularity of this cluster of control lines in more detail. This leads to the assumption that there is exactly one minterm which is associated with a cluster of control lines of one data path component. Note that this leads to an underestimation of the total switching.

2. Assume that the lines in a cluster are all active high. This condition can always be achieved by providing for a proper number of inversions at the data path end.

3. For the time being, we are ignoring the state counting process, which is the heart of a FSM. Later we will refine our model, and account for the energy required by the counting. Note that we still have some inaccuracy in our model due to the clustering of all the minterms for a data path component into one minterm. With each state transition in the state counting, only a subset of the cluster of control lines for a data path component switches. However, this information is too domain specific to allow for a general model.

Observe that due to the assumptions above, this minterm set has some nice properties. For any two minterms $\vec{u}$ and $\vec{v}$, $|\vec{u} - \vec{v}| \leq k_1$ and $|\vec{v} -$

$\vec{u}| \leq k_2$, for some constants k_1, k_2 depending on the FSM. Let k equal $max(k_1, k_2)$. This means that at most k new components can be activated and at most k currently active components can be deactivated in going to a next state. Note that k is, in a rough sense, the degree of parallelism or pipelining of the given data path.

In the case of a static design, going from state s_i to state s_j, on average k minterms will be turned off and k other minterms will be turned on. Thus, the average switching will be $2k$.

On the other hand, for a dynamic design the average switching equals $2m - 2\times$*(average size of an onset)*. Assuming that k is a very good approximation to the *onset* size, the average switching here is given by $2m-2k-2c$ for a very small c.

We could define $\rho = k/m$ to be *the degree of utilization / parallelism* for a FSM. Then, according to our previous discussion, a high value of ρ seems to favor the dynamic design, while a low value of ρ favors the static design style.

State Counting: As we mentioned earlier, state counting is an essential activity associated with a FSM. Typically, there is a microprogram for activating a data path unit consisting of several microcycles. For example, the first step in using an ALU in a dual bus architecture might be latching in the two operands from two buses. The current trend in Reduced Instruction [6] architectures seems to be to keep the number of microcycles, t, the same for each data unit. This, in turn, leads to uniform length machine instructions. In any case, we can assume that activating each data unit involves counting upto t, where t can be taken to be maximum of all data unit microprogram lengths. Due to the binary logic design, t will most likely be an integral power of 2. We now analyze the switching due to counting.

There are $h = \lceil log\, t \rceil$ output bits to count upto t. We assume the following sum of products form for the output bits of the state counter.

$$x_i(t_{n+1}) = \overline{x}_i(t_n)x_{i-1}(t_n)\ldots x_1(t_n) + x_i(t_n)\overline{x}_{i-1}(t_n) + \ldots + x_i(t_n)\overline{x}_1(t_n) \quad (3)$$

for all i, $1 \leq i \leq n$. The first product in this expression asserts the condition for toggling x_i from 0 to 1, that x_i is 0 and every other lower bit

is 1. Remaining terms give conditions for retaining x_i at 1, once it is 1. Note that for the ith output bit, x_i, there are i minterms in this expression. Also notice that the set of minterms for x_i is disjoint from the set of minterms for x_j for $i \neq j$.

Note that we could have worked with the Gray code counting rather than the lexicographic counting thus switching only one bit in going to the next state. However, as the reader can verify, it does not change the relative switching of two design styles.

In a static design, the least significant bit, x_1, toggles t times, x_2 toggles $t/2$ times, and x_i toggles $t/2^{i-1}$ times. We state the result for a static design as a theorem.

Theorem 5 *Let a static design PLA, designed with the logic expression stated above, count upto t. The total switching in counting from 0 to t is given by $t \log t$.*

PROOF: Let h equal $\lceil log\, t \rceil$. We refer to the complete enumeration of h-bit vectors. Some observations about this table are as follows. Each of the h columns has $t/2$ 0's and $t/2$ 1's. In the ith row (for $1 \leq i \leq h$), there are $t/2^i$ clusters of $2^{(i-1)}$ 1's.

We analyze the switching due to a 1-cluster in the ith column. Note that the first term in the R.H.S. of the Equation 3 is *on* only for the duration of the first 1 in the cluster. At that time all the other terms for x_i are off. The $(i-1)$st column determines when the second term for x_i is on. x_i is 1 everywhere in the 1-cluster. Thus for each 0-run in the $i-j$th column, $(j+1)$st term in Equation 3 stays on. Thus it is the number of 0-runs in the section of the $(i-j)$th column induced by this 1-cluster (restricted to this 1-cluster's rows), that determines the switching of $(j+1)$st term due to this 1-cluster. Summing this over $1 \leq j \leq i-1$ gives the switching due to one 1-cluster in the ith column, $Switch(i) = 2 + 2 + 4 + \ldots + 2^{i-1}$. This series sums up to 2^i. Since the number of 1-clusters is $t/2^i$, the total switching due to x_i is $(t/2^i)(2^i)$, which equals t. Thus the total switching for counting is given by $t \log t$.

□

How much switching energy do we need to count in a dynamic logic design? The answer is given by the following theorem.

Theorem 6 *Let a dynamic design PLA, designed with the logic expression in the Equation 3 above, count upto t. The total switching in counting from 0 to t is given by $\sim t \log^2 t$.*

PROOF: Each zero in a column contributes 2 to the total switching, 1 for *prechrge* and 1 for *evaluate*. Each of the $h = \log t$ columns has $t/2$ zeros. The switching due to zeros is given by $t(1 + 2 + 3 + \ldots + h)$, which equals $t \log t(\log t + 1)/2$. It turns out that the switching due to 1's is of the same order.

Let us estimate the switching due to one 1-cluster in the ith column. Note that the first term from the R.H.S. of the Equation 3 stays *on* only for the first 1 of the 1-cluster. Thus, every other time it is precharged and then discharged. The total switching of this term is given by $2(2^{i-1} - 1)$, where 2^{i-1} is the size of a 1-cluster. Every other term switches twice (precharge and discharge) for each zero in columns $i - 1$ through 1. This switching equals $(i - 1)(2\,2^{i-2})$. The total switching due to 1's is given by

$$\sum_{i=1}^{\log t} \frac{t}{2^i}[(2^i - 2) + (i - 1)2^{i-1}] .$$

This sum simplifies to the expression

$$t\left[\frac{\log^2 t + 3\log t}{4} - 2 + \frac{2}{t}\right] \sim t \log^2 t .$$

□

In most of the cases, the number of microcycles for any data unit will not exceed 32. With $t = 32$, the expression for static switching evaluates to 160 and it is 738 for dynamic switching. Thus even for the range of practical operation, the switching due to counting can change the balance in favor of static style, where dynamic style would otherwise have been a clear winner. The total switching with counting is given by following corollaries.

Corollary 1 *The average switching for a data path FSM designed in the static style is given by* $(k_1 + k_2)(t \log t)$, *where* k_1, k_2 *and* t *are as defined above.*

Corollary 2 *The average switching for a data path FSM designed in the dynamic style is given by* $(2m - k_1 - k_2)(t((\log t(\log t + 1)/2) + ((\log^2 t + 3\log t)/2 - 2 + 2/t)))$, *where* m, k_1, k_2 *and* t *are as defined above.*

5 Delay Analysis

Based on the assumption that the *power* $\times$ *delay* product remains constant, we expect a dynamic PLA to be faster than a static PLA. We show that it is indeed the case, for an *average* PLA. We explain later, what is meant by an *average* instance of a PLA.

Note that we are comparing the intrinsic differences between two design styles. The delay advantage gained by applying optimization techniques like folding is achievable in both the design styles. We assume that all the minterm lines have the same length in both the static and dynamic layouts, and they are routed in the same layer. In other words, all the minterms have the same capacitance and resistance. This assumption is valid for a PLA that has not gone through any layout optimizations like folding. We assume that the above mentioned conditions are valid for the output lines, as well. The techniques required to compare the output lines delay will be exactly the same as the ones used for comparing the minterm delay. Thus, we will demonstrate our results by minterm delay analysis.

It is clear from the delay model discussion that $0 \rightarrow 1$ transitions are prohibitively expensive in a static design, while they come for free in a dynamic design. Each $0 \rightarrow 1$ transition takes $\sim 5\nu$ delay, while a strength k $1 \rightarrow 0$ transition costs only ν/k. We will show that an average PLA instance has a lot of $0 \rightarrow 1$ transitions, hence proving our claim. The following assumptions informally define, what we mean by an *average instance* of a PLA. We assume that m equals n. Hence the minterm *onsets* come from the set $\{0,1\}^n$. We also assume

1. that there is no sharing of minterms, *i.e.*, no two inputs have the same minterm *onset*.

2. from a given input vector $\vec{x}$, it is equally likely to go to any other input vector $\vec{y}$, where $\vec{x}, \vec{y} \in \{0,1\}^n$.

Theorem 7 *A static PLA takes c times as long an asymptotic average time as a dynamic PLA, to evaluate its 'and plane', where c equals* ~ 5.

PROOF: We count the number of input pairs (s_i, s_j), such that there is at least one minterm switching from zero to one, in going from s_i to s_j, for all 2^{2n} such pairs. Let $C(n)$ denote this number for the *onsets* of size n. We write the following recurrence equation for C.

$$C(1) = 1 \; ; \; C(n) \; = \; 3\,C(n-1) \; + \; 2^{2n-2};$$

The term $2C(n-1)$ occurs due to the internal contributions of two groups of $n-1$ sized *onsets*. All the 2^{n-1} elements of the first group will have a $0 \rightarrow 1$ transition when going to any of the 2^{n-1} second group elements, since the first column has 2^{n-1} zeros followed by 2^{n-1} ones. This gives the term 2^{2n-2}. Another $C(n-1)$ factor comes along because of the interaction of the second group with the first one.

This recurrence has a solution in $C(n) \; = \; 4^n \; - \; 3^n$. Thus the number of such pairs grows exponentially in n. The average fraction of $0 \rightarrow 1$ transitions is given by $C(n)/2^{2n}$, which equals $1 \; - \; (3/4)^n$. Thus, the average delay is dominated by the $0 \rightarrow 1$ delay. The time it takes for the $1 \rightarrow 0$ transitions is the same in both the styles, since these transitions have the same strengths in both the design styles.

□

Note that in this delay model, the superiority of a dynamic design over a static design can be proved for any logic expression by showing that a majority of transitions are from zero to one.

6 Acknowledgements

I am thankful to my advisor, Larry Snyder, for many invaluable discussions and insights. I am grateful to Bill Beckett and Larry McMurchie for a careful reading of this paper. This work was funded in part by the Defence Advanced Research Projects Agency under contract number MDA903-85-K-0072.

References

[1] G. Bilardi, M. Pracchi, and F. P. Preparata. A Critique and an Appraisal of VLSI Models of Computation. In *Proceedings of CMU Conference on VLSI*, CMU, Computer Science Press, 1981.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston, Mass., 1984.

[3] G. De Micheli and A. Sangiovanni-Vincentelli. *PLEASURE: A Computer Program for Simple / Multiple Constrained / Unconstrained Folding of Programmable Logic Arrays.* Memorandum No. UCB/ERL M82/57, U.C. Berkeley, 1982.

[4] G. Hachtel, A.R. Newton, and A. Sangiovanni-Vincentelli. An Algorithm for Optimal PLA Folding. *IEEE Trans. on CAD of ICs and Systems*, April 1982.

[5] S. Ho, B. Jinks, T. Knight, J. Schaad, L. Snyder, A. Tyagi, and C. Yang. The Quarter Horse: A Case Study in Rapid Prototyping of a 32-bit Microprocessor Chip. In *IEEE Proceedings of the International Conference on Computer Design: VLSI in Computer*, IEEE Computer Society, 1985.

[6] M. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* Ph.D. Thesis, Computer Science Dept., U.C. Berkeley, 1983.

[7] G. Kissin. Functional Bounds on Switching Theory. In *Chapel Hill Conference on VLSI*, U.N.C., Chapel Hill, Computer Science Press, 1985.

[8] G. Kissin. Measuring Energy Consumption in VLSI Circuits: a Foundation. In *ACM Symposium on Theory of Computing*, ACM-SIGACT, 1982.

[9] C. Lutz. *Design of the Mosaic Processor.* Masters Thesis, Computer Science Dept., California Institute of Technology, 1984.

[10] W.D. Moeller and G. Sandweg. The Peripheral Processor PP4: A Highly Regular VLSI Processor. In *Proceedings of 11th International Symposium on Computer Architecture*, 1984.

[11] C.D. Thompson. Area-Time Complexity for VLSI. In *ACM Symposium on Theory of Computing*, ACM-SIGACT, 1979.

[12] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley Publishing Company, Reading, Mass., 1985.